A

PROJECT REPORT

ON

"Deep Neural Network Optimization"

SUBMITTED TO

SHIVAJI UNIVERSITY, KOLHAPUR

IN THE PARTIAL FULFILLMENT OF REQUIREMENT FOR THE AWARD OF DEGREE BACHELOR OF TECHNOLOGY IN COMPUTER SCIENCE AND ENGINEERING

SUBMITTED BY

MR. CHETAN PRAKASH BAVACHE	19UCS010
MR. PRANAV YASHWANT BHOKARE	19UCS014
MR. AKSHAY ANANDA CHOUGALE	19UCS022
MR. OMKAR RAJU DHANALE	19UCS030
MR. KISHOR SHASHIKANT HANGE	19UCS042

UNDER THE GUIDANCE OF

PROF. P. M. GAVALI

DKTE

Promoting Excellence in Teaching, Learning & Research

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DKTE SOCIETY'S TEXTILE AND ENGINEERING INSTITUTE, ICHALKARANJI

2022-23

D.K.T.E. SOCIETY'S

TEXTILE AND ENGINEERING INSTITUTE, ICHALKARANJI
(AN AUTONOUMOUS INSTITUTE)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

DKTE
Promoting Excellence in
Teaching, Learning & Research

CERTIFICATE

This is to certify that, project work entitled

"Deep Neural Network Optimization"

is a bonafide record of project work carried out in this college by

MR. CHETAN PRAKASH BAVACHE	**19UCS010**
MR. PRANAV YASHWANT BHOKARE	**19UCS014**
MR. AKSHAY ANANDA CHOUGALE	**19UCS022**
MR. OMKAR RAJU DHANALE	**19UCS030**
MR. KISHOR SHASHIKANT HANGE	**19UCS042**

is in the partial fulfilment of award of degree Bachelor in Engineering in Computer Science & Engineering prescribed by Shivaji University, Kolhapur for the academic year 2016-2017.

PROF. P. M. GAVALI
(PROJECT GUIDE)

PROF.(DR.) D.V. KODAVADE
(HOD CSE DEPT.)

PROF.(DR.) L.S. ADMUTHE
(DIRECTOR)

EXAMINER: _________________

____

DECLARATION

We hereby declare that, the project work report entitled "Deep Neural Network Optimization" which is being submitted to D.K.T.E. Society's Textile and Engineering Institute Ichalkaranji, affiliated to Shivaji University, Kolhapur is in partial fulfilment of degree B.Tech.(CSE). It is a bonafide report of the work carried out by us. The material contained in this report has not been submitted to any university or institution for the award of any degree. Further, we declare that we have not violated any of the provisions under Copyright and Piracy / Cyber / IPR Act amended from time to time.

MR. CHETAN PRAKASH BAVACHE	19UCS010
MR. PRANAV YASHWANT BHOKARE	19UCS014
MR. AKSHAY ANANDA CHOUAGLE	19UCS022
MR. OMKAR RAJU DHANALE	19UCS030
MR. KISHOR SHASHIKANT HANGE	19UCS042

ACKNOWLEDGEMENT

With great pleasure we wish to express our deep sense of gratitude to Prof. P. M. Gavali for his valuable guidance, support and encouragement in completion of this project report.

Also, we would like to take opportunity to thank our head of department Dr. D. V. Kodavade for his co-operation in preparing this project report.

We feel gratified to record our cordial thanks to other staff members of Computer Science and Engineering Department for their support, help and assistance which they extended as and when required.

Thank you,

MR. CHETAN PRAKASH BAVACHE	19UCS010
MR. PRANAV YASHWANT BHOKARE	19UCS014
MR. AKSHAY ANANDA CHOUAGLE	19UCS022
MR. OMKAR RAJU DHANALE	19UCS030
MR. KISHOR SHASHIKANT HANGE	19UCS042

ABSTRACT

Optimizations for large-scale deep learning models encompass various techniques that aim to enhance both model efficiency and accuracy. One such technique involves optimizing the file format used to store the models. Deep learning models are typically saved in file formats like TensorFlow's Saved Model or ONNX (Open Neural Network Exchange), which offer flexibility and portability across different frameworks. However, these formats may not be optimized for efficient inference on edge devices due to their larger size and overhead. To address this, one effective technique is to convert the models into more compact and efficient formats like TensorFlow Lite (TFLite) or Core ML. These formats are specifically designed for resource-constrained devices and offer specialized operations and runtime environments tailored to mobile and embedded platforms. By converting models to these formats, we can significantly reduce the model size, enabling faster inference and improved resource utilization on edge devices without compromising accuracy. Additionally, these formats provide compatibility with various edge device frameworks, simplifying deployment and integration processes.

Another approach to optimizing large-scale deep learning models is by utilizing TVM (Tensor Virtual Machine), an open-source compiler stack that aims to provide efficient deployment of deep learning models on a variety of hardware platforms. TVM optimizes models by performing various transformations, such as operator fusion, kernel specialization, and hardware-specific code generation. Operator fusion involves combining multiple operations into a single computation kernel, reducing memory access and improving cache utilization. Kernel specialization generates customized kernels tailored to the target hardware, exploiting specific hardware features for optimized execution. Moreover, TVM incorporates auto-tuning techniques, which explore a wide range of hardware configurations and schedules to find the best combination for a given model and target device. By leveraging TVM's capabilities, deep learning models can be optimized for specific hardware architectures, including CPUs, GPUs, and specialized accelerators, making them more suitable for efficient deployment on edge devices. The automatic optimization process offered by TVM minimizes manual effort and maximizes performance, enabling models to leverage the full potential of the underlying hardware while maintaining accuracy.

Quantization is another powerful technique for optimizing deep neural networks, particularly for deployment on edge devices with limited computational resources. It involves reducing the precision of model weights and activations from floating-point to lower bit representations, such as 8-bit integers. Quantization reduces the memory footprint and computational requirements of the model, enabling more efficient inference on edge devices. While quantization introduces some loss in model accuracy due to the reduced precision, various techniques have been developed to mitigate this impact. Quantization-aware training involves training the models with the awareness of the quantization process, allowing the model to adapt to the lower precision during the training phase. This approach helps in preserving the accuracy of the quantized model by reducing the quantization-induced errors.

Another technique is post-training quantization, where a pre-trained model is quantized without requiring retraining. This technique allows flexibility in choosing different quantization schemes and can be applied to models trained with high precision. Furthermore, advancements in hardware support for quantized operations, such as Intel's VNNI (Vector Neural Network Instructions) or NVIDIA's Tensor Cores, further enhance the speed and efficiency of quantized models on supported hardware platforms. These hardware optimizations enable efficient execution of quantized models, minimizing the performance gap between quantized and full-precision models.

In summary, optimizations for large-scale deep learning models involve various techniques to improve both model efficiency and accuracy. Optimizations using file format conversion, such as leveraging TFLite or Core ML, reduce the model size and enable faster inference on edge devices. TVM optimization, with its operator fusion, kernel specialization, and auto-tuning capabilities, optimizes models for specific hardware architectures, maximizing performance while maintaining accuracy. Quantization, combined with techniques like quantization-aware training and post-training quantization, reduces model precision to achieve higher inference efficiency on edge devices. Hardware advancements further enhance the speed and efficiency of quantized models. By employing these techniques, deep learning models can be optimized for efficient and accurate deployment on edge devices, enabling a wide range of applications in resource-constrained environments.

INDEX

1. Introduction

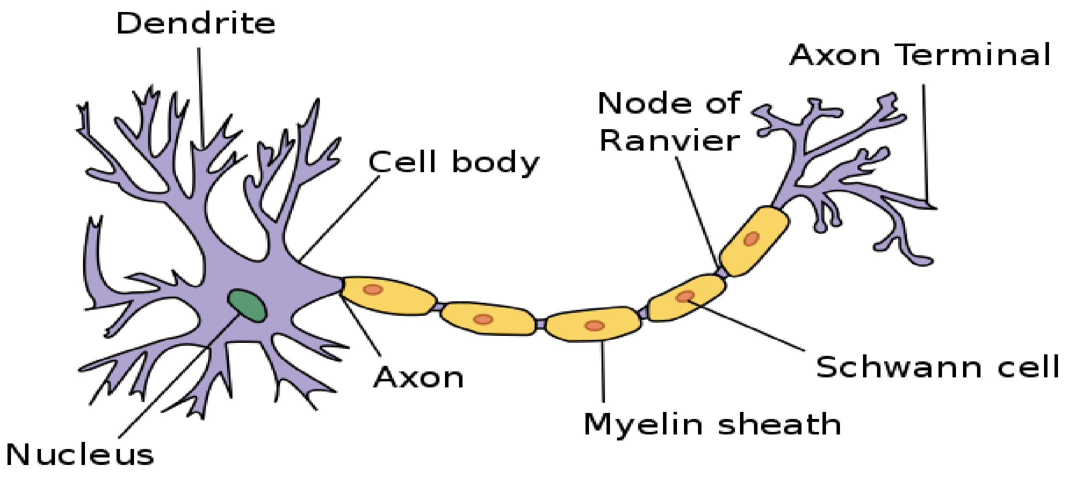


Fig.1 Neuron structure in human brain

The inspiration behind neural networks is the remarkable human brain. Artificial neural networks (ANNs) are an attempt to simulate the network of neurons that make up the human brain, enabling computers to learn and make decisions in a human-like manner. ANNs are created by programming regular computers to behave as interconnected brain cells. Similar to the human brain, neural networks consist of a large number of interconnected elements that mimic neurons. These elements, also known as artificial neurons or nodes, receive inputs, perform computations, and produce outputs. By leveraging the interconnectedness and parallel processing capability of these artificial neurons, neural networks can process vast amounts of data, recognize complex patterns, and make predictions or decisions. This bio-inspired approach has revolutionized fields such as computer vision, natural language processing, and speech recognition, enabling machines to perform tasks that were once thought to be exclusive to human intelligence.

Optimizers play a crucial role in training neural networks, although many may be using them without realizing they are applying optimization techniques. Optimizers are algorithms or methods used to change the attributes of a neural network, such as weights and learning rates, to reduce the losses and improve performance. An optimization problem can be categorized as either linear or nonlinear. Linear optimization problems involve objective functions and constraints that are linear. These problems are relatively simpler and easier to solve, as all linear functions are convex. On the other hand, nonlinear optimization problems are more complex, as the decision space is nonconvex and the objective function may have multiple local optima. Finding the global solution, which is the optimal solution among all possible solutions, becomes challenging in such cases. Various optimization algorithms, such as gradient descent, stochastic gradient descent, and Adam, are employed to navigate the complex parameter space of neural networks and find the optimal values for improved performance. These optimization techniques not only enhance the accuracy of neural networks but also accelerate the training process, allowing for the development of more efficient and accurate models.

Deep neural networks have shown great potential in a wide range of applications, including computer vision, speech recognition, and natural language processing. However, their deployment on resource-constrained edge devices has been challenging due to the computational requirements and memory constraints of these devices. To address this, researchers and engineers are developing techniques to enable the use of deep neural networks on edge devices. One approach involves model compression, where the large and complex models are compressed or distilled into smaller and more efficient versions without significant loss in accuracy. Techniques such as pruning, quantization, and knowledge distillation are used to reduce the model size and computational requirements, making them more suitable for edge devices. Additionally, specialized hardware architectures and accelerators are being designed to efficiently execute deep neural networks on edge devices, leveraging techniques like parallel processing and optimized memory access. These advancements in hardware and model optimization enable the deployment of deep neural networks on edge devices, bringing capabilities such as face recognition, speech recognition, and enhanced AI features directly to the devices, improving their capabilities and user experiences. With the increasing demand for edge computing and the proliferation of IoT devices, the use of deep neural networks on edge devices holds great promise for various applications, including smart homes, autonomous vehicles, and healthcare monitoring systems.

The deployment of deep neural networks on edge devices has the potential to revolutionize various industries and domains. In the healthcare sector, edge devices equipped with deep neural networks can enable real-time monitoring of patients, allowing for early detection of medical conditions and personalized treatment recommendations. This has the potential to greatly improve patient outcomes and reduce the burden on healthcare systems. Furthermore, in the field of agriculture, deep neural networks deployed on edge devices can analyse sensor data from farms to optimize irrigation, fertilization, and pest control, leading to increased crop yields and resource efficiency. Edge devices with embedded deep neural networks can also enhance security systems by enabling real-time video analysis for surveillance and intruder detection, making our homes and businesses safer. Moreover, the integration of deep neural networks into edge devices can lead to advancements in autonomous systems. Autonomous vehicles, for example, can benefit greatly from the deployment of deep neural networks on board. These networks can analyse sensor data in real-time, enabling the vehicle to detect and respond to traffic conditions, pedestrians, and obstacles, enhancing safety and enabling more efficient and reliable transportation. Similarly, in the field of robotics, deep neural networks can empower robots to perceive and interact with their environment, enabling them to perform complex tasks autonomously. This opens up possibilities for applications such as warehouse automation, medical robotics, and home assistance robots.

The use of deep neural networks on edge devices is not limited to specialized industries alone. It can also have a profound impact on our everyday lives. For instance, edge devices with embedded deep neural networks can enhance personal digital assistants by enabling natural language processing and understanding. This allows for more accurate and context-aware interactions with virtual assistants, making them more intuitive and helpful. Additionally, smart home devices can benefit from deep neural networks by understanding user preferences, optimizing energy consumption, and providing personalized experiences. From adjusting the lighting and temperature to managing home security systems, deep neural networks integrated into edge devices can make our living spaces more intelligent and responsive to our needs.

Furthermore, deep neural networks integrated into edge devices can revolutionize the field of healthcare. These devices can enable real-time analysis of patient data, such as vital signs and symptoms, allowing for early detection of diseases and personalized treatment recommendations. By leveraging deep neural networks, edge devices can assist healthcare professionals in making accurate diagnoses and improving patient outcomes. Additionally, wearable devices equipped with deep neural networks can monitor individuals' health parameters, providing proactive health monitoring and timely alerts for potential medical emergencies.

In the realm of finance, deep neural networks on edge devices can enhance fraud detection systems. By analysing transaction data in real-time, these networks can identify suspicious patterns and anomalies, enabling early fraud detection and prevention. This not only helps protect individuals and businesses from financial losses but also strengthens overall security in the financial sector.

Transportation and logistics can also benefit from the deployment of deep neural networks on edge devices. Autonomous vehicles, for example, can leverage deep neural networks for object recognition, path planning, and decision-making, enabling safer and more efficient transportation. Similarly, logistics companies can use deep neural networks for real-time tracking, optimizing delivery routes, and improving supply chain management, leading to cost savings and improved customer satisfaction.

The entertainment industry can also harness the power of deep neural networks on edge devices. Smart devices equipped with deep neural networks can provide personalized recommendations for movies, music, and other forms of entertainment based on users' preferences and behaviour patterns. This creates a more immersive and tailored entertainment experience for individuals.

Overall, the integration of deep neural networks into edge devices has the potential to revolutionize various aspects of our everyday lives. From personalized assistance and efficient energy management to improved healthcare and enhanced security, deep neural networks on edge devices can transform our homes, workplaces, and communities, making them more intelligent, efficient, and user-centric. As advancements in optimization techniques and hardware capabilities continue, we can expect to see even more innovative applications of deep neural networks on edge devices, creating a truly connected and intelligent world.

The deployment of deep neural networks on edge devices not only holds immense potential for various industries and domains but also presents new challenges that require innovative solutions. For example, the limited memory and processing power of edge devices may result in high inference latency and reduced accuracy. To address this, researchers are exploring new techniques such as federated learning, which allows edge devices to learn collaboratively while preserving privacy and security. In this approach, instead of sending raw data to the cloud for training, models are trained directly on the edge devices by aggregating updates from multiple devices, allowing for better utilization of local data and reducing communication costs. Federated learning can also help in overcoming data privacy concerns, as data never leaves the edge device, and only the updates are shared with the central server.

Another challenge in deploying deep neural networks on edge devices is the need for efficient energy consumption. The energy requirements of deep neural networks can be significant, making them impractical for battery-powered edge devices. To address this, researchers are developing new hardware architectures that are optimized for deep learning workloads, such as domain-specific accelerators. These specialized chips are designed to perform only specific tasks, such as matrix multiplication and convolution, making them more energy-efficient than general-purpose CPUs. Additionally, techniques such as model pruning, quantization, and knowledge distillation can be used to reduce the size of deep neural networks, making them more suitable for edge devices with limited memory and processing power.

Despite these challenges, the deployment of deep neural networks on edge devices is expected to grow rapidly, fuelled by the increasing demand for intelligent and connected devices. The market for edge computing is expected to grow at a compound annual growth rate of 28.3% from 2020 to 2027, according to a report by Grand View Research. With the proliferation of IoT devices and the growing need for real-time processing and decision-making, the use of deep neural networks on edge devices is set to transform various industries and domains, including healthcare, manufacturing, transportation, and smart cities. By leveraging the power of deep learning, edge devices can become more intelligent and responsive, enhancing our everyday lives and driving innovation.

In summary, the deployment of deep neural networks on edge devices holds immense potential for various industries and domains. By bringing intelligence and decision-making capabilities closer to the data source, edge devices equipped with deep neural networks can overcome the limitations of cloud-based processing, enabling real-time analysis, reduced latency, improved privacy, and enhanced user experiences. As optimization techniques continue to advance and hardware capabilities improve, we can expect to see a wide range of innovative applications and solutions that harness the power of deep neural networks on edge devices, transforming the way we interact with technology and unlocking new possibilities for a smarter and more connected world.

a. Goals and Objectives:

1. The primary objective of this project is to optimize the network complexity of deep neural networks, striking a balance between model efficiency and accuracy.
   By reducing the network complexity, we aim to enable the deployment of deep neural networks on resource-constrained devices, such as edge devices and mobile devices, without compromising their performance.
   The project will explore techniques such as network pruning, weight sharing, and model compression to minimize computational requirements and memory footprint while maintaining or improving the accuracy of the deep neural network. Through careful selection of network architectures, hyperparameter tuning, and rigorous evaluation, we strive to achieve an optimized deep neural network that can efficiently capture and represent the underlying patterns in the data, leading to improved efficiency, reduced latency, and enhanced user experiences in edge computing scenarios.
2. Utilize model to decrease the amount of time needed for model calculation.
   Another objective of this project is to leverage advanced techniques and methodologies to significantly reduce the time required for model calculation in deep neural networks, enhancing their efficiency and responsiveness.
   By optimizing the model calculation process, we aim to accelerate the inference speed of deep neural networks, enabling real-time decision-making and analysis in applications where time-critical responses are crucial.
   Through the exploration of model optimization techniques, parallel computing, and hardware acceleration, we seek to leverage the full potential of modern computing resources to minimize the computational overhead and maximize the throughput of deep neural network calculations.
3. Utilize an improved model to save memory.
   By optimizing the model architecture and parameters, we aim to reduce the memory footprint of deep neural networks without compromising their performance and accuracy, enabling their deployment on memory-limited devices such as edge devices and mobile platforms.
   Through techniques such as model compression, parameter sharing, and network pruning, we will explore ways to decrease the number of parameters and activations required by the model, resulting in a more compact representation that requires less memory for storage and computation.
   The project will involve researching and developing novel algorithms and methodologies that strike a balance between memory efficiency and model performance, with the ultimate goal of enabling the deployment of deep neural networks on devices with limited memory resources, expanding the scope of their applications in fields such as Internet of Things (IoT), wearable devices, and embedded systems.

b. Aim:

- Design and develop solution to optimize the Deep Neural Network.

The aim of this project is to design and develop an innovative solution to optimize Deep Neural Networks (DNNs) and improve their performance and efficiency. The proposed solution will involve a multi-faceted approach that encompasses various aspects of DNN optimization, including model architecture, parameter tuning, regularization techniques, and data pre-processing. By carefully analysing the specific challenges and requirements of the DNN model, we will identify areas for improvement and develop novel strategies to enhance its overall performance. Through rigorous experimentation and iterative refinement, we aim to create a robust and efficient solution that maximizes the accuracy and computational efficiency of DNNs, making them more suitable for resource-constrained environments such as edge devices and mobile platforms. The ultimate objective is to empower businesses and industries with optimized DNN models that can unlock new possibilities and deliver superior results in applications such as computer vision, natural language processing, and speech recognition.

c. Scope and limitation:

Scope

The use of neural networks offers superior accuracy compared to general machine learning algorithms. However, one of the major challenges is deploying these neural networks on edge devices due to their high computation time and memory usage. The objective of this project is to address these challenges by optimizing the deep neural network to make it suitable for deployment on edge devices such as cameras, IoT devices, and other resource-constrained platforms.

The main focus of this project is software optimization of the deep neural network. By applying various optimization techniques, we aim to reduce the computational complexity and memory requirements of the neural network without compromising its accuracy. The goal is to achieve similar accuracy levels as the original model while significantly improving its efficiency and suitability for edge devices.

It is important to note that the accuracy of the optimized model may vary within a considerable range. However, our objective is to ensure that the optimized model maintains a comparable level of accuracy to the original model, even though minor variations may occur. This allows us to strike a balance between efficiency and accuracy, enabling the successful deployment of neural networks on edge devices.

While the project primarily focuses on software optimization, it does not specifically target hardware optimization. However, by optimizing the software aspect of the deep neural network, we aim to enhance its overall performance, which can subsequently benefit from advancements in hardware technologies as well. The optimized model can leverage hardware improvements in edge devices to further enhance its speed and efficiency, providing a comprehensive solution for deploying neural networks on edge devices.

Limitations:

1. Specific versions are necessary to optimise the model.
2. Dynamic models whose input shape or even the model itself may change in execution are not fully supported by the TVM compiler. TVM only supports static models.

d. Timeline of Project:

Activity	Jul-22	Aug-22	Sep-22	Oct-22	Nov-22	Dec-22	Jan-23	Feb-23	Mar-23	Apr-23
Define Project	Identified the problem & investigate it									
Research Review		Tried to learn more about project through research paper								
Requirement Gathering			All requirement must be determined for base design							
Document Synopsis				Documented all requirement and design pattern						
Model Creation					Created DNN model for optimization					
Model Conversions						H5 model converted into PB,Optimized PB, TFLite and Onnx Models				
Model Optimization and Evaluation							All model feed to TVM compiler & optimized using fusion technique			
Model Testing								Performance of Optimized model tested against various input models		
Research Paper									Observations and analysis were used to create the research paper.	
Final Report Writing and Review										Based on the output, a project report was produced

e. Project Management Plan:

1. Milestone List

The figure below shows the important milestones in deep neural optimisation. This chart contains significant project milestones such as project phase completion or gate review. Smaller milestones that are not included in this chart but are in the project timeline and work breakdown structure (WBS) are possible. If there are any schedule delays that could affect a milestone or delivery date, the project lead must be alerted as soon as possible so that proactive efforts can be made to avoid date slips. The project lead will notify the project team of any approved changes to these milestones or dates.

Milestone	Description	Start Date	End Date	Duration	Priority
Define Project	Identify the problem and investigate it	01/07/22	14/07/22	14 Days	High
Research Overview	Try to learn more about project through research papers	15/07/22	29/07/22	14 Days	High
Requirement Gathering	All the requirement for DNN optimization must be determined to base diagram	30/07/22	12/08/22	14 Days	High
Document Synopsis	Document all requirement and design pattern	13/08/22	27/08/22	14 Days	Medium
Model Creation	Created DNN model for optimization	28/08/22	04/09/22	7 Days	Medium
Model Conversions	H5 model converted into PB, Optimized PB, Tflite and Onnx Models	05/09/22	03/10/22	28 Days	High
Model Optimization and evaluation	All model feed to TVM compiler & optimized using fusion technique	01/01/23	28/01/23	28 Days	High
Model Testing	Performance of Optimized model tested against various input models	29/01/23	28/02/23	28 Days	High

Research Paper	Observations and analysis were used to create the research paper.	01/03/23	21/03/23	21 Days	Medium
Final Report Writing and Review	Based on the output, a project report was produced	22/03/23	13/04/23	14 Days	Medium

f. Project Cost:

COCOMO Model:

In this project, the Cost Estimation based on COCOMO (Constructive Cost Model) the formula for this Model is follows:

Effort = Constant × (Size) scale factor× Effort Multiplier

– Effort in terms of person-months

– Constant: 2.45 in 1998 based on Organic Mode

– Size: Estimated Size in KLOC

– Scale Factor: combined process factors

– Effort Multiplier (EM): combined effort factors

Functional Point Table:

The function point range in between 1-10

Conversion of Functional point to Lines of Code (LOC)

Total function points = 6

Estimated Size:

The basic COCOMO equations take the form

Effort Applied (E) = ab (KLOC) bb [man-months]

Development Time (D) = cb (Effort Applied) db [months]

People required (P) = Effort Applied / Development Time [count]

Where, KLOC is the estimated number of delivered lines (expressed in thousands) of code for a project. The coefficients ab, bb, cb and db are given in the following table.

Software Project	a	b	c	d
Organic	2.3	0.75	1.01	0.35
Semidetached	1.1	0.8	1.01	0.8
Embedded	3.6	0.31	1.01	0.04

Semidetached Mode:

Effort Applied (E) = 1.1*(600) ^0.8 = 2047.57058

Development Time (D) = 1.1*(2047.57058) ^0.8 = 490.23832

People Required (P) = 2024.57058 / 490.23832

= 4.17 People

= 4 People

2. Background Study and Literature Overview:

1. Hanan Hussain [1] has conducted research on the challenges of deploying neural networks on edge devices, driven by the increasing demand for edge-level training due to the rise in AI applications and the need for privacy and security. To address these challenges, researchers are exploring various technologies such as federated learning, incremental training, and accelerator arrays. However, deploying deep neural network (DNN) models on edge devices can be challenging due to their complex nature and lack of consideration for implementation complexity during their design, as they primarily focus on maximizing accuracy.

   To tackle this issue, hardware-DNN model co-design is being explored as a potential solution. The co-design approach encompasses two main strategies. Firstly, reducing the precision of operators and operations through techniques like quantization, weight sharing, and reduced bandwidth can help optimize the model's computational requirements and memory usage. Secondly, reducing the number of operations and model size through compression and pruning techniques can further enhance the efficiency of the deployed DNN model on edge devices.

   In addition to hardware-DNN model co-design, software optimizations that are aware of the underlying hardware are being investigated. These optimizations aim to leverage the specific characteristics of edge devices to improve the performance and efficiency of DNN models. However, further research and effort are needed to fully explore and exploit the potential of hardware-aware software optimizations for edge devices.

   Overall, the research by Hanan Hussain highlights the challenges of deploying neural networks on edge devices and proposes hardware-DNN model co-design, reduced precision, and reduced number of operations as potential solutions. The exploration of hardware-aware software optimizations holds promise but requires additional investigation and development to succeed in optimizing DNN models for edge devices.

2. Arvind Krishnamurthy [4] has introduced a research paper focusing on the TVM compiler, a group deep neural network optimizer. The paper presents an end-to-end compilation stack that aims to address fundamental optimization challenges in deep learning across various hardware back-ends. The proposed solution revolves around tensorization and compiler-driven latency hiding to efficiently target accelerators.

   The TVM compiler offers a generic solution that enables seamless optimization for deep learning models across different hardware platforms. By leveraging tensorization techniques, the compiler optimizes the tensor computations involved in deep neural networks, maximizing their efficiency and performance on specific hardware architectures. Additionally, the compiler-driven latency hiding approach helps minimize the impact of latency in accelerating the execution of deep learning models.

   One of the key contributions of the research paper is the automation of the end-to-end optimization process. Traditionally, optimizing deep learning models has been a labour-intensive and specialized task that requires significant manual effort. However, the proposed system automates the optimization process, reducing the need for manual intervention and expertise. This automation streamlines the optimization pipeline and makes it more accessible to researchers and practitioners, enabling faster development and deployment of optimized deep neural network models.

In summary, Arvind Krishnamurthy's research paper on the TVM compiler highlights the significance of an end-to-end compilation stack in addressing optimization challenges in deep learning for diverse hardware back-ends. By incorporating tensorization and compiler-driven latency hiding, the TVM compiler provides a powerful tool for optimizing deep neural network models. The automation of the optimization process enhances productivity and accessibility, accelerating the development of efficient deep learning solutions.

3. In this research paper, the author Qingchao Shen [3] address the increasing use of deep learning compilers to generate optimized code for deep learning models on specific hardware. While these compilers can improve runtime performance, they can also introduce bugs that can lead to unexpected behaviours and potentially catastrophic consequences in critical systems. The authors emphasize that deep learning models differ from traditional imperative programs in terms of their implicit program logic, requiring a re-evaluation of bug characteristics in the context of deep learning compilers.

   To investigate this issue, the authors conduct a systematic study of DL compiler bugs, analyzing 603 bugs from three popular DL compilers: TVM, Glow, and nGraph. They examine the root causes, symptoms, and stages of occurrence during compilation for these bugs. Through their analysis, the authors identify 12 key findings and provide valuable guidelines for future research on DL compiler bug detection and debugging.
   One notable finding is that a significant portion (nearly 20%) of DL compiler bugs are related to types, particularly tensor types. This insight leads to the development of new mutation operators, such as adding type casts for tensors to enable implicit type conversion in subsequent tensor computations. These operators facilitate the detection of type-related bugs. Additionally, the authors introduce TVMfuzz as a proof-of-concept application based on their findings. TVMfuzz generates new tests using the original TVM test suite and successfully exposes eight previously missed bugs.

   Overall, this research provides a comprehensive analysis of DL compiler bugs and offers valuable insights for improving bug detection and debugging in deep learning compilers. By addressing the unique characteristics of deep learning models and their compilation process, the authors contribute to the advancement of reliable and robust deep learning systems.

3. Requirement Analysis:

a. Requirement analysis and gathering

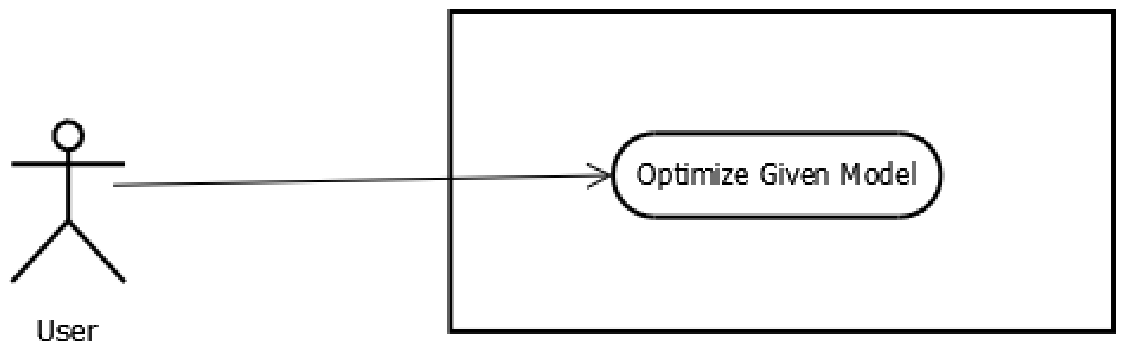
- **Hardware requirements:**

Number	Description	Alternatives (If available)
1	PC with 2 GB hard-disk and 8 GB RAM	Not Applicable
2	Integrated Graphics Card	NVIDIA

- **Software requirements:**

Number	Description	Alternatives (If available)
1	Windows 7/10/11 Or Ubuntu	Not Applicable
2	Python 3.0 or above	Not Applicable
3	TensorFlow 2.0v or above	Not Applicable
4	TF Lite	Not Applicable

b. Use case Diagram:



In this use case diagram, the focus is on the optimization of deep neural networks, and it showcases the interactions between a single user and the optimization system. The system offers several key use cases to facilitate the optimization process.

Train Model: The first use case is "Train Model," which involves training a deep neural network model using a given dataset and a set of hyperparameters. The user can provide the system with the dataset they wish to use for training and specify the hyperparameters that govern the model's learning process. The system then executes the training procedure and generates a trained model.

Validate: The next use case is "Validate," which aims to evaluate the performance of the trained model. The user can select a holdout dataset that is separate from the training data. The system then applies the trained model to this dataset and measures its performance metrics, such as accuracy or loss. This validation process helps ensure that the trained model generalizes well to unseen data.

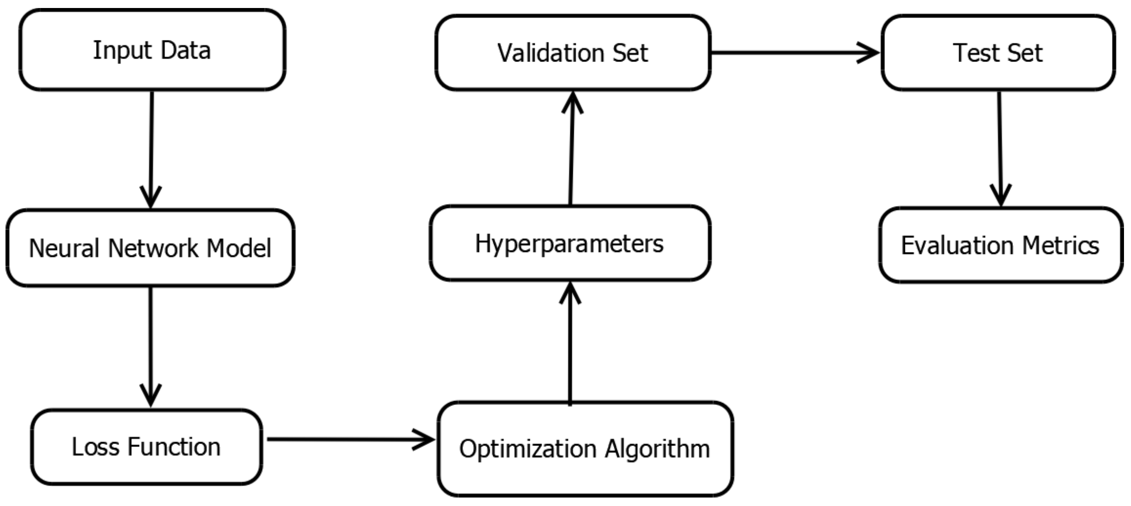
Test Model: Another important use case is "Test Model," which focuses on evaluating the performance of the trained model on a new, unseen dataset. The user provides the system with this test dataset, and the system applies the trained model to make predictions or classifications. The results are then analyzed to assess the model's performance in real-world scenarios.

Optimize: Lastly, the "Optimize" use case involves fine-tuning the hyperparameters of the model to enhance its performance further. The user can interact with the system to specify the hyperparameters they want to optimize and set optimization criteria. The system utilizes various techniques, such as grid search or evolutionary algorithms, to explore different hyperparameter configurations and identify the optimal set of values that yield the best performance for the model.

Overall, this use case diagram illustrates the main steps involved in the optimization of deep neural networks, providing users with an interface to interact with the system and perform essential tasks like training, validation, testing, and hyperparameter optimization.

4. System Design

a. Architecture Design:



1. Input Data: The architecture diagram starts with the input data, which could be images, text, audio, or any other form of data suitable for the problem at hand.

2. Neural Network Model: The neural network model is the core component of the optimization process. It consists of multiple layers of interconnected nodes (neurons) that perform computations on the input data. The model typically includes layers such as convolutional layers, fully connected layers, recurrent layers, etc., depending on the specific problem and network architecture.

3. Loss Function: The loss function measures the discrepancy between the predicted output of the neural network and the true labels or target values. It quantifies the model's performance and is used as a basis for optimization.

4. Optimization Algorithm: The optimization algorithm determines how the neural network model is trained to minimize the loss function. It includes techniques such as gradient descent, stochastic gradient descent, Adam, RMSprop, etc. These algorithms update the model's parameters (weights and biases) iteratively to improve its performance.

5. Hyperparameters: Hyperparameters are configuration settings that control the behavior of the neural network and the optimization process. They include learning rate, batch size, number of layers, activation functions, regularization techniques, etc. Hyperparameter tuning is an essential part of deep neural network optimization.

6. Validation Set: A portion of the available labeled data is often set aside as a validation set. It is used to evaluate the model's performance during training and guide the selection of optimal hyperparameters. The validation set helps prevent overfitting and allows for early stopping if the model's performance deteriorates.

7. Test Set: Once the model training is complete, a separate test set, consisting of unseen data, is used to assess the model's performance on new instances. It provides an unbiased evaluation of the model's ability to generalize to real-world data.

8. Evaluation Metrics: Evaluation metrics such as accuracy, precision, recall, F1 score, or mean squared error are used to quantify the performance of the optimized model. These metrics provide quantitative measures of how well the model performs on the task at hand.

b. Algorithmic description of each module

1.Start

2.Conversion of H5 format to PB format

    1.Save Model in H5.
    2.Import TensorFlow library.
    3. Load a PB File
    4.Convert the model into graph.
    5.Convert the graph into PB.
    6.Save model.

3.Optimize PB format model.

    1. Import TensorFlow library.
    2. Load a PB file.
    3. Optimize the PB model using transform_graph function.
    4. Save Model in PB format.

4.Convert PB format to tflite format

    1. Import TensorFlow libraries.
    2. Load a PB file.
    3. Convert the PB format file in tflite format using TFLiteConverter.
    4. Save Model in tflite format.

5.Convert tflite format to ONNX format

    1. Import onnxruntime library.
    2. Load a tflite file.
    3. Convert the tflite format to onnx file format using tflite2onnx.
    4. Save Model in ONNX format.

6.Convert PB format to ONNX format

    1. Import onnxruntime libraries.
    2. Load a PB file.
    3. Convert the tflite format to onnx file format using tflite2onnx.
    4. Save Model in ONNX format.

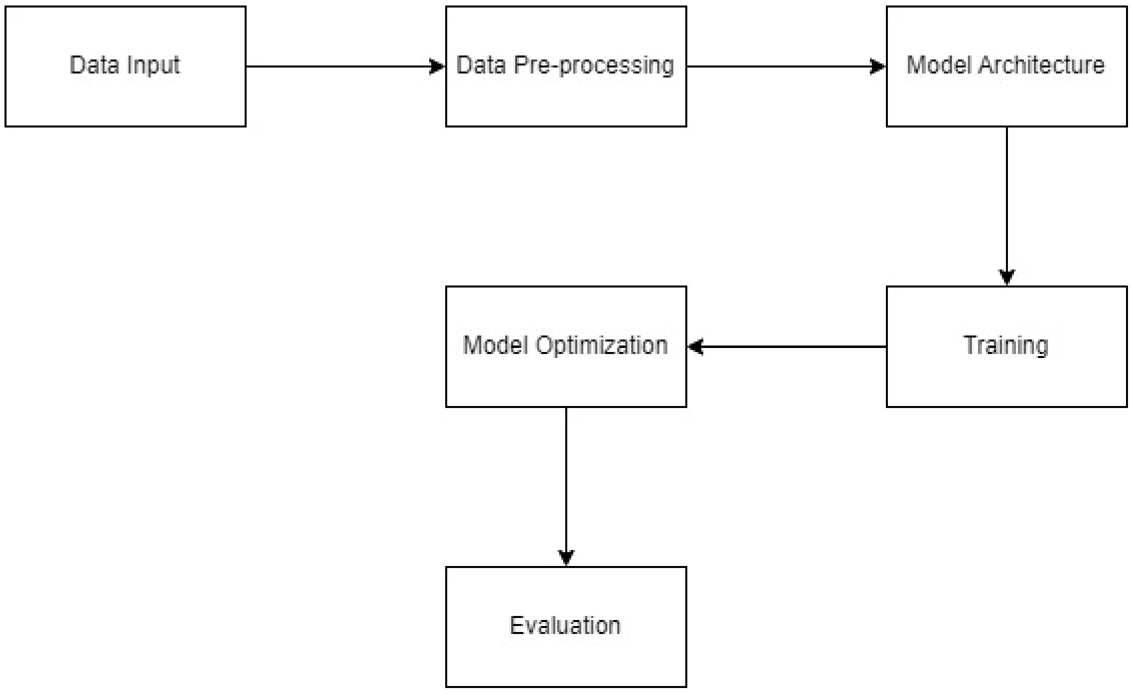
7.Optimize given model using TVM compiler

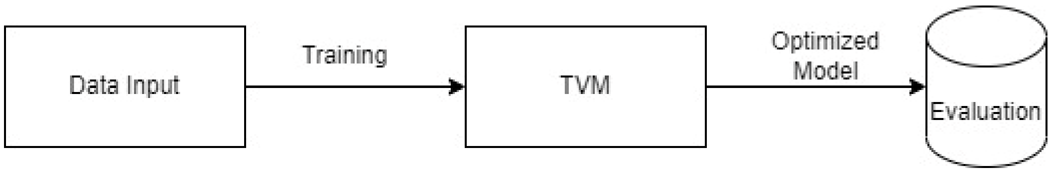
   1. Compiling an ONNX Model on the TVM Runtime.
   2. Running the Model from The Compiled Module with TVMC.
   3. Evaluating the model.

8. End.

c. System Modelling:

1. Dataflow Diagram:





Data Input: The initial step in the process is to enter the data into the system. The data can be in various forms such as images, text, audio, or any other type of input that the neural network is designed to process. This data serves as the raw input for the subsequent stages of the process.

Data Pre-processing: Once the data is entered into the system, it undergoes pre-processing. Data pre-processing involves several tasks such as data normalization, feature extraction, and data cleaning. Normalization ensures that the data is scaled to a standard range, feature extraction extracts relevant features from the data, and data cleaning involves handling missing values, removing outliers, and dealing with noise. The goal of data pre-processing is to prepare the data in a suitable format that can be effectively utilized by the neural network during training.

Model Architecture Design: After the data has been pre-processed, the next step is to design the architecture of the neural network. This involves making decisions about the type of network architecture to be used, such as convolutional neural networks (CNNs) for image data or recurrent neural networks (RNNs) for sequential data. Additionally, the number of layers, the activation functions used in each layer, and the number of neurons in each layer need to be determined. The architecture design is crucial as it defines how information flows through the network and influences the network's ability to learn and make accurate predictions.

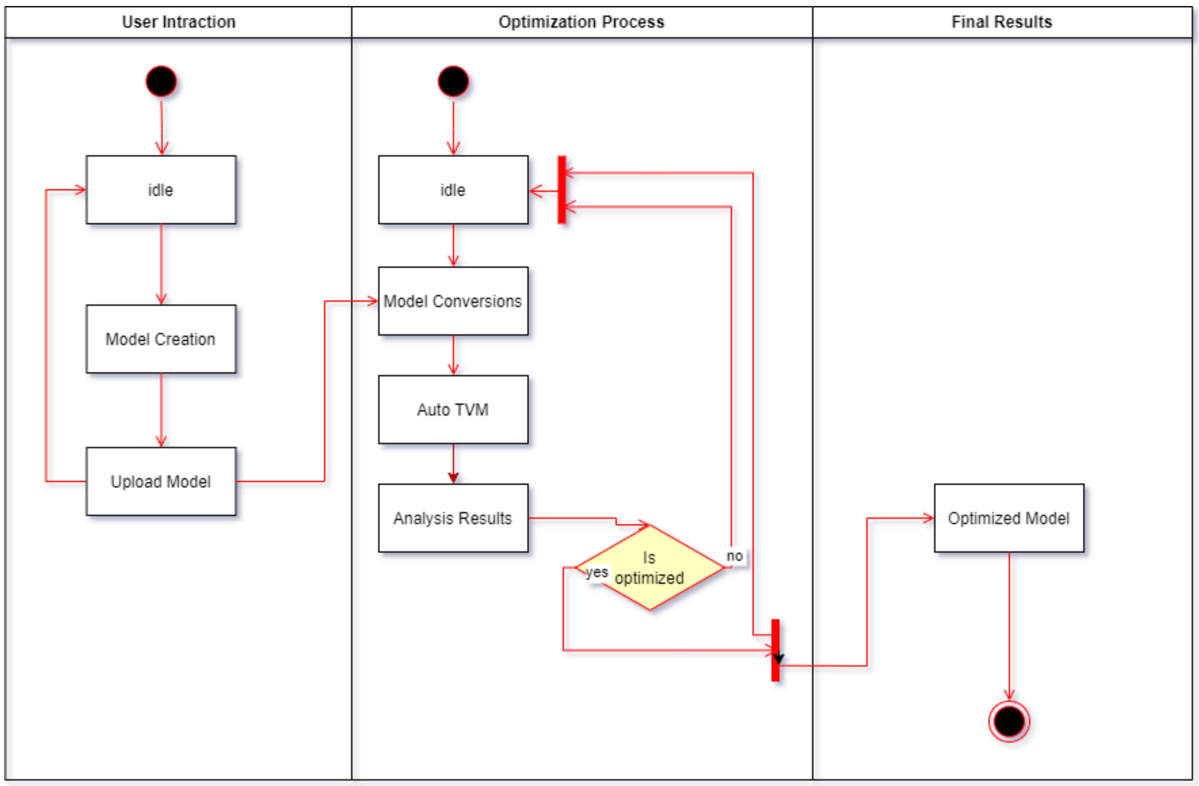
Training: Once the model architecture has been designed, the neural network is trained using the pre-processed data. During training, the network adjusts its weights iteratively to minimize the discrepancy between the expected output and the actual output. This is typically done through an optimization algorithm, such as stochastic gradient descent (SGD), which updates the weights based on the error calculated during the forward and backward propagation. The training process continues for multiple epochs until the network learns to make accurate predictions on the training data.

Model Optimization: After the initial training, the model can be further optimized to improve its performance. Various techniques can be employed for optimization. Ensemble learning methods, such as bagging or boosting, can be used to combine multiple models and improve prediction accuracy. Regularization techniques, such as dropout or L1/L2 regularization, can be applied to prevent overfitting and improve generalization. Hyperparameter tuning involves adjusting the settings that are not learned during training, such as learning rate, batch size, or regularization strength, to find the optimal configuration for the model. Model optimization aims to enhance the model's performance and robustness.

Evaluation: Once the model has been optimized, its performance is evaluated using a separate test dataset. This evaluation is important to assess how well the model generalizes to unseen data. Various performance metrics such as accuracy, precision, recall, and F1 score can be calculated to measure the model's effectiveness. Evaluation provides insights into the model's strengths and weaknesses, and helps in identifying potential areas for improvement or further optimization.

In summary, the process of training a neural network involves data input, data pre-processing, model architecture design, training the network, model optimization, and evaluation of the model's performance. Each step contributes to building a robust and accurate neural network model that can effectively process input data and make reliable predictions.

2. Activity Diagram:



1. The first activity in the project is to train the model using a suitable artificial neural network (ANN). This involves selecting an appropriate ANN architecture based on the problem at hand, such as feedforward neural networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs), or transformers. The model is trained using labeled data, where the network learns to make predictions by adjusting its weights through an optimization algorithm like backpropagation.

2. After training the model, it needs to be stored in a file format for future use. One commonly used file format is the H5 format, which is particularly associated with Keras (now officially part of TensorFlow). The H5 format is used to save the weights and architecture of the model. It is a data-oriented format and is less programmatic compared to other formats like the Protocol Buffer (PB). Saving the model in H5 format allows for easier reusability and compatibility with Keras and TensorFlow.

3. In certain scenarios, there may be a need to convert the H5 format model to the PB format. PB format, short for Protocol Buffer format, is a way to store structured data, including neural networks. PB is an open-source project overseen by Google. The conversion is typically done using a script called freeze_graph.py, which takes the H5 format model and converts it into a PB format model. This conversion can be useful for specific deployment requirements or when using certain tools or frameworks that rely on the PB format.

4. TensorFlow Lite (TFLite) is a set of tools designed to enable on-device machine learning on mobile, embedded, and edge devices. It allows developers to run their models efficiently on devices with limited resources such as memory and computation power. To deploy the deep neural network model on such devices, it is often necessary to convert the PB format model to the TFLite format. TFLite also performs optimization on the model by default, tailored to the constraints of mobile and edge devices.

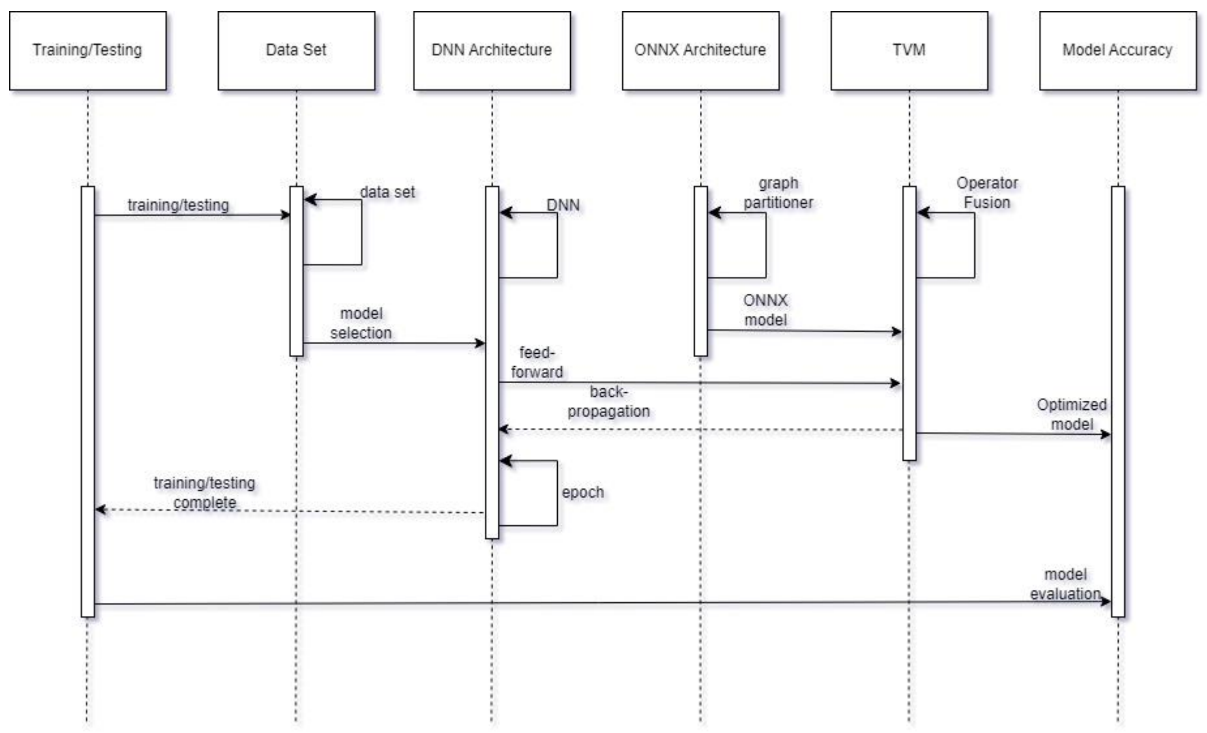
5. The next step involves converting the TFLite format model to the ONNX (Open Neural Network Exchange) format. ONNX is a framework-agnostic format that aims to provide a common language for describing machine learning models. The goal is to make it easier to deploy models in production environments by decoupling the learning framework from the deployment framework. Converting the TFLite model to ONNX allows for interoperability and flexibility when working with different deep learning frameworks.

6. To further optimize the obtained model, the TVM (Tensor Virtual Machine) compiler can be employed. TVM is a compiler that focuses on providing performance portability for deep learning workloads across diverse hardware backends. It offers graph-level and operator-level optimizations, enabling high-level operator fusion, mapping to various hardware primitives, and efficient memory latency hiding. By leveraging TVM, the model can be fine-tuned and optimized for specific hardware architectures, leading to improved performance.

7. After obtaining the optimized model, it is essential to evaluate its performance. Evaluation scripts in Python can be used to assess the accuracy and other relevant metrics of the optimized model. By comparing the performance of different models, including the original and optimized versions, it becomes possible to determine which model yields the best results. If the optimized model demonstrates considerable improvements in accuracy or other desired metrics, it can be chosen as the preferred model and replace the original model for deployment.

In summary, the project involves training an artificial neural network, storing the model in suitable file formats such as H5 and PB, converting between different formats like PB to TFLite to ONNX, optimizing the model using tools like TVM, evaluating the performance of different models, and selecting the most optimized model for deployment. Each step in the process contributes to improving the efficiency, portability, and performance of the neural network model in various deployment scenarios.

3. Sequence Diagram:



Data pre-processing: Data pre-processing is a crucial step in preparing raw data for training a neural network. It involves several tasks to transform the data into a suitable format for effective utilization by the network during training. In addition to feature extraction, data enrichment, data cleansing, and normalization, data pre-processing may also include handling missing values, dealing with outliers, and encoding categorical variables. These processes aim to enhance the quality and relevance of the data, reduce noise, and ensure that the data is in a consistent and meaningful format.

Deep Neural network architecture: The deep neural network architecture determines how the network is organized and structured. It involves making decisions about the number and types of layers in the network, the activation functions used within each layer, and the connections between the layers. The architecture plays a significant role in defining how information flows through the network and affects the network's ability to learn and make accurate predictions. Different architectures, such as feedforward neural networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformers, are designed for specific tasks and data types.

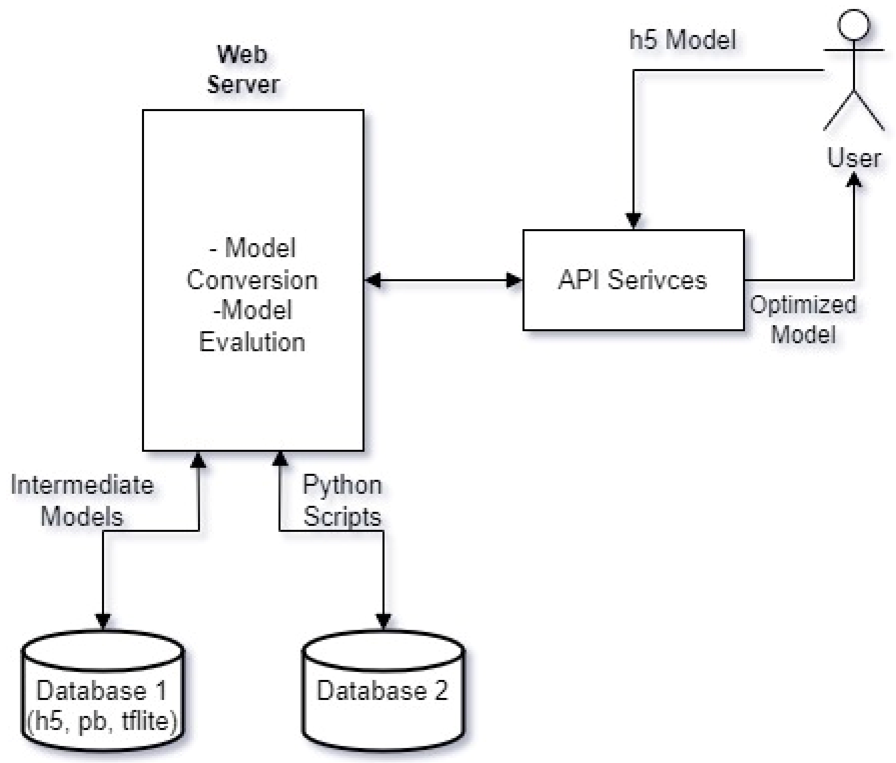
Hyperparameter tuning: Hyperparameter tuning involves selecting optimal values for the hyperparameters that govern the behavior of the neural network. Hyperparameters are settings that are not learned during training but affect the learning process. Examples of hyperparameters include the learning rate, batch size, number of hidden units, regularization strength, and dropout rate. Tuning these parameters is essential to optimize the performance of the neural network and improve its ability to generalize to new data. It is typically done through a search process, where different combinations of hyperparameters are evaluated using a validation dataset.

Validation: Validation is the process of evaluating the trained neural network's performance on a separate validation dataset. The validation dataset is distinct from the training dataset and serves as a measure of how well the network generalizes to unseen data. By evaluating performance metrics such as accuracy, precision, recall, or mean squared error on the validation dataset, it is possible to assess the network's effectiveness and identify potential issues like overfitting or underfitting. Validation helps in understanding how the network will likely perform on new, unseen data.

Optimization: Optimization involves making adjustments to the hyperparameters or neural network architecture to improve its performance on the validation dataset. It is an iterative process where different configurations of hyperparameters are tested and evaluated. Techniques such as grid search and random search can be used to systematically explore the hyperparameter space, while more advanced methods like Bayesian optimization or genetic algorithms can help in finding the best combination of settings more efficiently. The goal of optimization is to refine the neural network's configuration and improve its predictive capabilities on unseen data.

In summary, the data pre-processing component prepares the raw data, the neural network architecture defines the structure of the network, hyperparameter tuning selects optimal settings, validation evaluates performance, and optimization refines the network's configuration to improve its predictive capabilities. Overall, the process of training a neural network involves iteratively going through data pre-processing, defining the neural network architecture, tuning hyperparameters, evaluating performance through validation, and optimizing the network configuration. This iterative approach helps in building a robust and accurate neural network model for a given task.

4. Deployment Diagram:



5. Implementation

a. Environmental Setting for Running the Project

- Environment setting for software:
  1. Python 2.0 or above
  2. Anaconda 2021 or above
  3. Tensorflow 2.1 or above
  4. TVM compiler 0.11.1
- Environment setting for hardware:
  1. PC with CPU of intel i3 or above
  2. RAM of minimum 2GB
  3. Disk space minimum of 4GB
  4. NVIDIA Graphics card (Optional)

b. Detailed Description of methods:

1. **Data Collection**

   Data requirement process is carried out by the administrator to ensure the identified requirements are relevant and feasible. He can optimize the given model using this project. He can achieve almost 6 times faster model than original model. It has the authority to manage step by step execution of a proposed model. The data taken for training is from keras library.

2. **Model Training**

   Model is created by use of data collected in previous step. A 2D convolutional model is created by use of dataset to recognize the digits from image. The model uses 'relu' activation function for input layer and 'softmax' activation function for output layer. Adam optimizer is used while compiling the model.

3. **Store Model in H5 format**

   Different file formats with different characteristics, both used by tensorflow to save models (.h5 specifically by keras)

   Used originally by keras to save models (keras is now officially part of tensorflow). It is less general and more "data-oriented", less programmatic than pb. To store the weights and architecture of the model, we must use h5 format in keras and save model in H5 format.

4. **Conversion of H5 format model to PB format**

   It is a way to store some structured data (in this case a neural network), project is open source and currently overviewed by Google.

   First line of optimization in our project is to convert the H5 format model in Pb format using freeze_graph.py script. Converting the model allows you to easily deploy it in these environments without any compatibility issues. PB format provides a standardized representation of the model that can be loaded and used by various TensorFlow APIs and tools. The conversion process essentially involves transforming the model from one file format (H5) to another (PB) without altering the underlying model itself. The conversion ensures compatibility with different deployment environments, tools, and platforms that expect models to be in the PB format. This allows for seamless integration, deployment, and inference on various devices or frameworks that support the PB format.

   Tensorflow has a core library to convert the h5 to protobuf format. For this conversion the h5 model is converted into saved_model and after that it is saved into protocol buffer(pb) using the SavedModel format is converted to the PB format using the TensorFlow Lite converter() method. After the conversion, the PB model is obtained as a byte array (tflite_model), which can be saved to disk with the ".pb" file extension.

5. **Conversion of PB format model to tflite format**

   In this step, the TensorFlow Lite converter (tf.lite.TFLiteConverter) is used to convert the PB model to the TFLite format. The converter takes the PB model as input and performs the conversion. After the conversion, the TFLite model is obtained as a byte array (tflite_model), which can be saved to disk with the ".tflite" file extension. The code above saves the TFLite model to the specified location.

TensorFlow Lite performs operator fusion during the conversion process. It combines multiple consecutive operations or layers into a single fused operation. By fusing operations together, unnecessary memory transfers and intermediate computations are eliminated, resulting in improved computational efficiency and reduced memory overhead during inference. Once the model is converted to the TFLite format, you can deploy it on mobile or edge devices that support TensorFlow Lite. The TFLite format is specifically optimized for efficient inference on resource-constrained devices, providing faster execution and reduced model size. Tensorflow lite is a set of tools that enables on-device machine learning by helping developers run their models on mobile, embedded, and edge devices. As tflite framework is used to train the deep neural network in small devices (devices having less memory and computation).

6. **Conversion of tflite model to ONNX format**

ONNX aims at providing a common language any machine learning framework can use to describe its models. The first scenario is to make it easier to deploy a machine learning model in production. An ONNX interpreter (or runtime) can be specifically implemented and optimized for this task in the environment where it is deployed. With ONNX, it is possible to build a unique process to deploy a model in production and independent from the learning framework used to build the model.

In this step we load the tflite model using the interpreter method of the tflite. Onnx uses the prepare method from the onnx backend library. In this step, the onnx_tf.backend.prepare() function is used to convert the TFLite model to the ONNX format. The function takes the TFLite model (interpreter) as input and prepares it for conversion. The resulting ONNX model is then exported and saved to the specified location using the export() method.

So, we have to convert the tflite to onnx format.

7. **Optimize Obtained model using TVM compiler**

TVM provides a wide range of optimization techniques, including auto-tuning, kernel fusion, and operator specialization, which can explore further to achieve optimal performance on target hardware. We propose TVM, a compiler that exposes graph-level and operator-level optimizations to provide performance portability to deep learning workloads across diverse hardware back-ends. TVM explores opportunities for parallel execution across operators in the computation graph. It identifies independent subgraphs and parallelizes their execution, enabling concurrent execution and utilizing multiple cores or accelerators for faster computation. TVM leverages the knowledge of the target hardware's Instruction Set Architecture (ISA) to generate hardware-specific code. It utilizes the instruction-level parallelism, vectorization capabilities, and specialized instructions available on the target architecture to optimize the implementation of operators. For example, it may generate SIMD instructions like AVX or NEON to perform parallel computations on vectorized data. TVM applies a range of compiler optimizations to the code generated for specific operators. These optimizations include loop unrolling, loop fusion, loop reordering, constant folding, and register allocation. By optimizing the control flow and memory accesses within operators, TVM reduces computational overhead and improves performance.

8. **Evaluate All Models**

   Evaluate all obtained model using evaluation scripts in python and decide which one is most optimized model. Check its accuracy is considerable or not, if the accuracy is in considerable range, then replace the original model with optimized model.

c. Implementation Details:

1. Creating model & generating graph using netron.ai.

   The process of testing deep neural network optimization begins with creating a model, typically using popular deep learning frameworks like TensorFlow or PyTorch. Once the model is trained and ready, it can be exported and visualized using tools like Netron.ai. Netron.ai allows researchers and developers to inspect the model's architecture, layer connections, and parameters through a graphical representation. This visualization aids in understanding the structure of the model and helps identify potential areas for optimization.

2. Conversion of model into ONNX & TFLITE model.

   To further optimize the deep neural network model, it can be converted into different formats. One commonly used format is the Open Neural Network Exchange (ONNX), an open standard that allows models to be interchanged between various deep learning frameworks. Conversion to ONNX format enables compatibility with different frameworks and tools, facilitating experimentation and optimization.

   Additionally, the model can be converted into TensorFlow Lite (TFLite) format, which is specifically designed for deployment on resource-constrained edge devices. TFLite models are lightweight and optimized for efficient execution on devices with limited computational capabilities. This conversion ensures that the model can be effectively deployed on edge devices, without compromising performance or accuracy.

3. Convert TFLITE model into ONNX & generate graph of both models.

   Once the model is in TFLite format, it can be further converted back to ONNX format if required. This conversion allows researchers to leverage the benefits of both formats, utilizing the compatibility and optimization features of TFLite while also accessing the flexibility and interoperability of ONNX. Generating graphs for both the TFLite and ONNX models using tools like Netron.ai helps visualize the network structure and gain insights into the optimized representations.

4. Both the ONNX models feed into TVM compiler and generate graph.

   The next step in the testing process involves feeding both ONNX models into the TVM (Tensor Virtual Machine) compiler. TVM is a compiler stack that optimizes deep learning models for efficient execution on various hardware platforms. By utilizing TVM, researchers can leverage the compiler's capabilities to optimize the models further, ensuring they are both performant and resource-efficient. The TVM compiler optimizes the computational graph of the models, applying techniques such as operator fusion, tensorization, and hardware-specific code generation.

5. Compare all the 5 graphs & pick a model with best performance in time & space having same accuracy as original model.
   Once all the optimized graphs are generated, researchers can compare them to identify the model with the best performance in terms of both time and space efficiency. This

evaluation involves analyzing factors such as inference speed, memory footprint, and computational requirements. The selected model should demonstrate comparable accuracy to the original model while exhibiting improved performance in terms of inference time and memory utilization.

By going through these steps of creating the model, converting to different formats, generating graphs, utilizing the TVM compiler, and comparing the optimized models, researchers can find the most optimized deep neural network model. This model strikes a balance between accuracy and efficiency, making it suitable for deployment on edge devices or resource-constrained environments, where computational resources and memory limitations are a concern.

6. Testing

Testing of deep neural network optimization involves evaluating the effectiveness and efficiency of various optimization techniques and hyperparameters in training a neural network. This can include assessing the model's ability to learn and generalize from the training data, as well as its performance on new, unseen data.

Here are some common approaches to testing deep neural network optimization:

Cross-validation: One common approach in testing deep neural network optimization is cross-validation. This technique involves dividing the dataset into multiple subsets or "folds." The model is then trained on each fold while being validated on the remaining folds. By performing cross-validation, researchers can assess the model's ability to generalize to new data and ensure that it does not rely too heavily on specific subsets of the data.

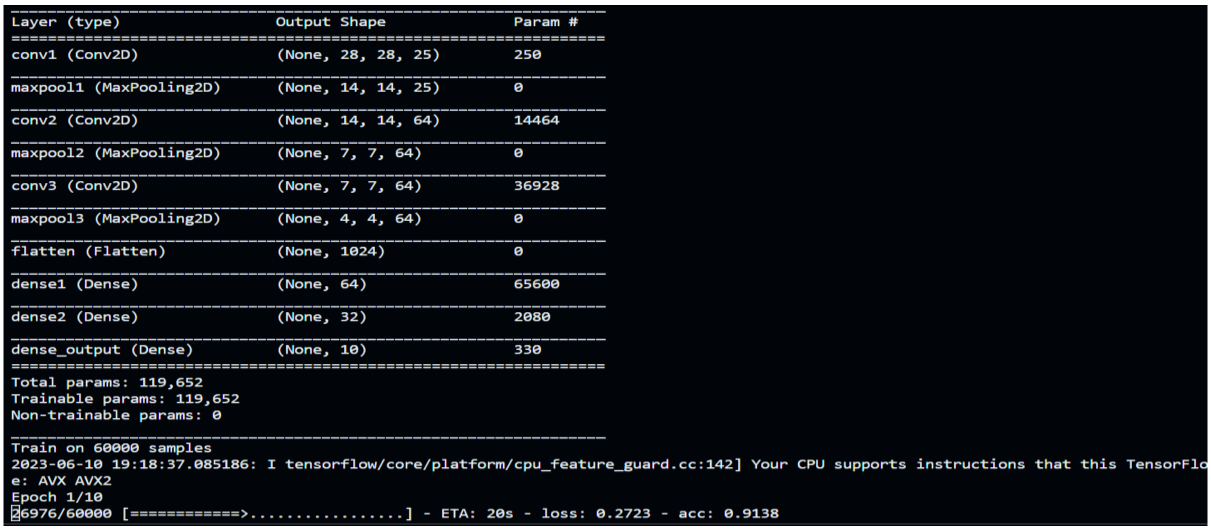
Hyperparameter tuning: Hyperparameter tuning is another essential aspect of testing deep neural network optimization. Hyperparameters, such as the learning rate, batch size, and number of layers, significantly influence the model's performance. Researchers systematically adjust these hyperparameters to find the optimal configuration that yields the best results on the validation set. This iterative process allows them to fine-tune the model and enhance its overall performance.

Early stopping: Early stopping is a technique often employed during training to prevent overfitting. Overfitting occurs when the model becomes too specialized in the training data and fails to generalize well to new data. By monitoring the validation loss, researchers can stop the training process when they observe an increase in validation loss, indicating that the model's performance on new data is starting to deteriorate. This prevents the model from being overly tuned to the training data and promotes better generalization.

Regularization techniques: Regularization techniques are crucial in testing deep neural network optimization to combat overfitting. Methods such as dropout and weight decay introduce constraints to the model during training, helping to prevent it from memorizing the training data too precisely. Dropout randomly "drops out" units (neurons) during training, forcing the model to learn more robust and generalizable representations. Weight decay adds a penalty term to the loss function, discouraging large weight values and encouraging simpler models.

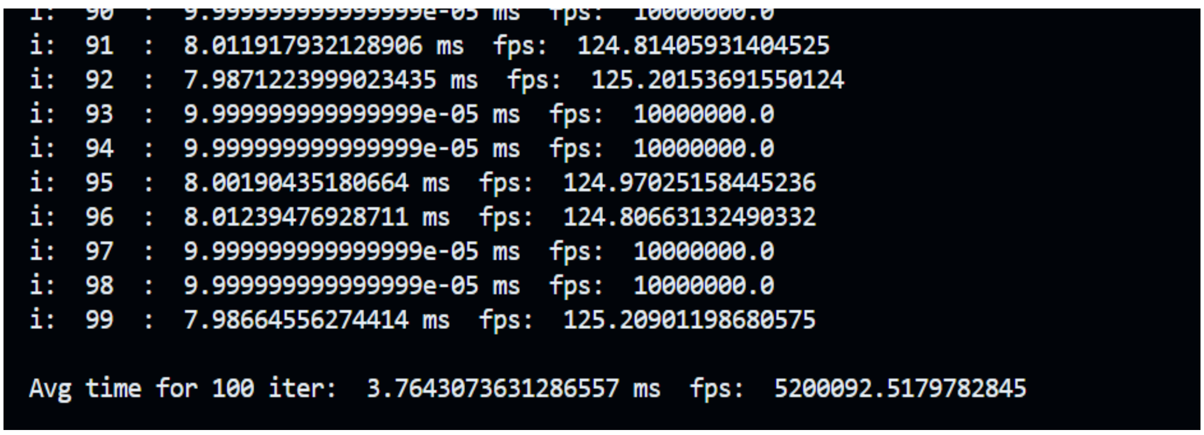
Testing on new data: Once the model has been optimized and trained on the training data, it is vital to evaluate its performance on new, unseen data. Testing on new data helps researchers validate the model's ability to generalize to real-world scenarios. By assessing its performance on unseen data, they can gain insights into its real-world applicability and ensure that it performs well outside the training context.

Once the model has been optimized and trained on the training data, it is vital to evaluate its performance on new, unseen data. Testing on new data helps researchers validate the model's ability to generalize to real-world scenarios. By assessing its performance on unseen data, they can gain insights into its real-world applicability and ensure that it performs well outside the training context.



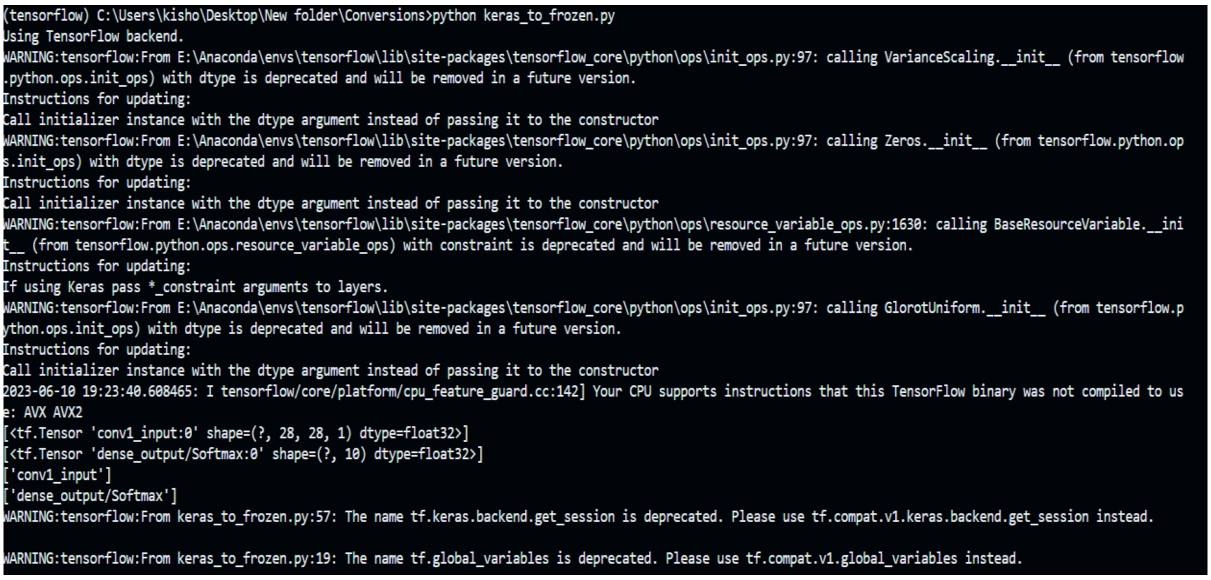
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1 (Conv2D)               (None, 28, 28, 25)        250
_________________________________________________________________
maxpool1 (MaxPooling2D)      (None, 14, 14, 25)        0
_________________________________________________________________
conv2 (Conv2D)               (None, 14, 14, 64)        14464
_________________________________________________________________
maxpool2 (MaxPooling2D)      (None, 7, 7, 64)          0
_________________________________________________________________
conv3 (Conv2D)               (None, 7, 7, 64)          36928
_________________________________________________________________
maxpool3 (MaxPooling2D)      (None, 4, 4, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense1 (Dense)               (None, 64)                65600
_________________________________________________________________
dense2 (Dense)               (None, 32)                2080
_________________________________________________________________
dense_output (Dense)         (None, 10)                330
=================================================================
Total params: 119,652
Trainable params: 119,652
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples
2023-06-10 19:18:37.085186: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlo
e: AVX AVX2
Epoch 1/10
26976/60000 [===========>................] - ETA: 20s - loss: 0.2723 - acc: 0.9138
```

Fig 1: Training Model



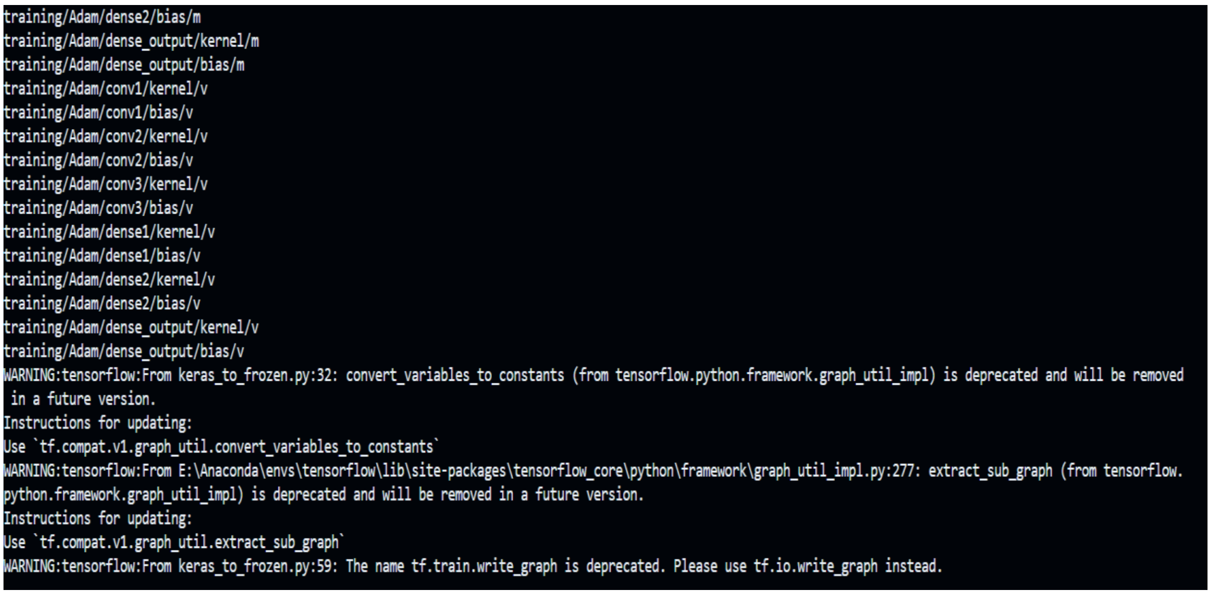
```
i:  90  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  91  :  8.011917932128906 ms  fps:  124.81405931404525
i:  92  :  7.9871223999023435 ms  fps:  125.20153691550124
i:  93  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  94  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  95  :  8.00190435180664 ms  fps:  124.97025158445236
i:  96  :  8.01239476928711 ms  fps:  124.80663132490332
i:  97  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  98  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  99  :  7.98664556274414 ms  fps:  125.20901198680575

Avg time for 100 iter:  3.7643073631286557 ms  fps:  5200092.5179782845
```

Fig 2: Model Evalution



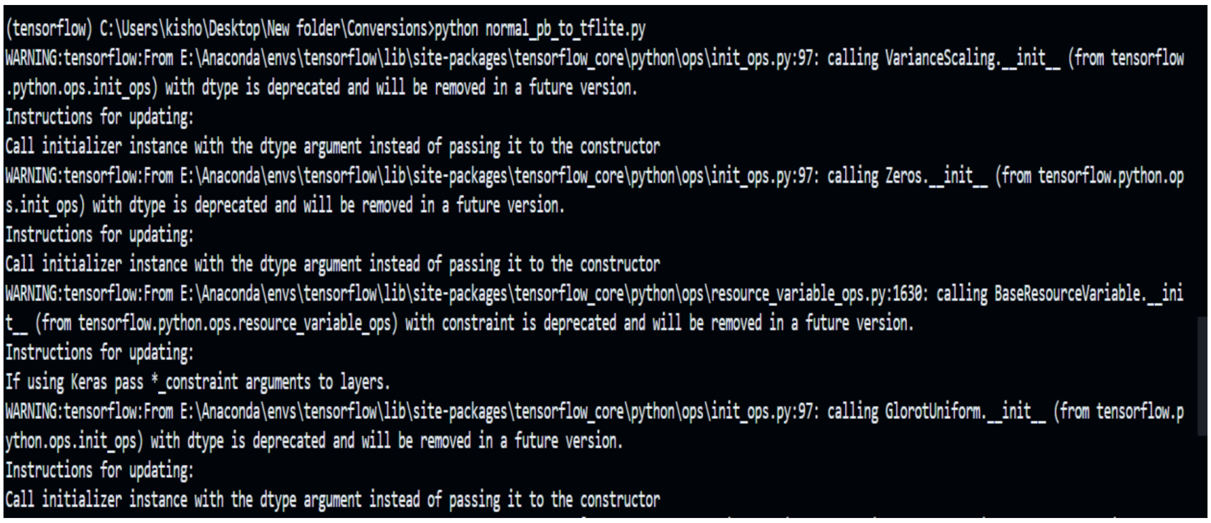
```
(tensorflow) C:\Users\kisho\Desktop\New folder\Conversions>python keras_to_frozen.py
Using TensorFlow backend.
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling VarianceScaling.__init__ (from tensorflow
.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling Zeros.__init__ (from tensorflow.python.op
s.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__ini
t__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling GlorotUniform.__init__ (from tensorflow.p
ython.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
2023-06-10 19:23:40.608465: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlow binary was not compiled to us
e: AVX AVX2
[<tf.Tensor 'conv1_input:0' shape=(?, 28, 28, 1) dtype=float32>]
[<tf.Tensor 'dense_output/Softmax:0' shape=(?, 10) dtype=float32>]
['conv1_input']
['dense_output/Softmax']
WARNING:tensorflow:From keras_to_frozen.py:57: The name tf.keras.backend.get_session is deprecated. Please use tf.compat.v1.keras.backend.get_session instead.

WARNING:tensorflow:From keras_to_frozen.py:19: The name tf.global_variables is deprecated. Please use tf.compat.v1.global_variables instead.
```

Fig 3: Conversion Of Keras to Frozen Graph



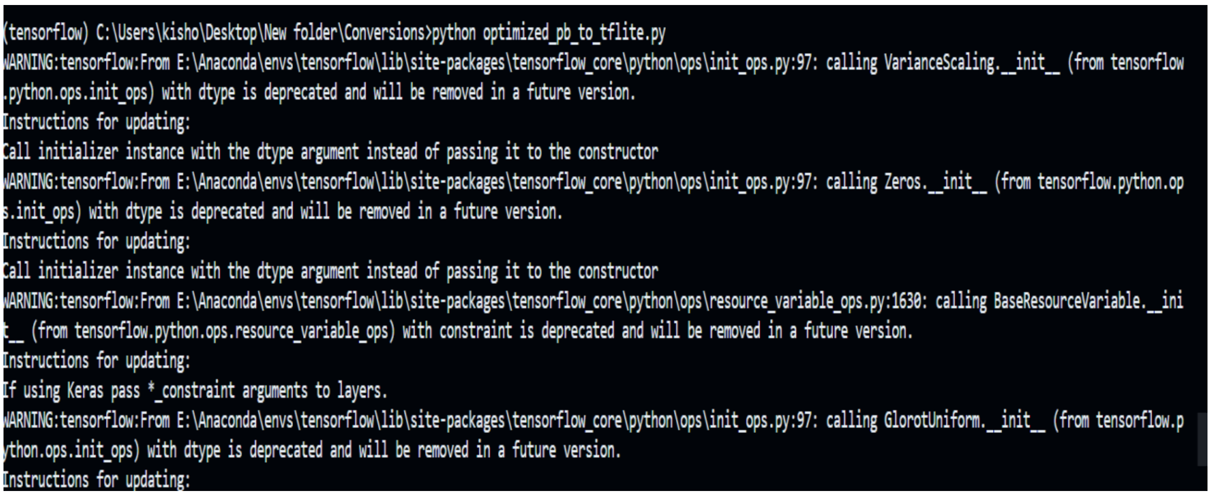
```
training/Adam/dense2/bias/m
training/Adam/dense_output/kernel/m
training/Adam/dense_output/bias/m
training/Adam/conv1/kernel/v
training/Adam/conv1/bias/v
training/Adam/conv2/kernel/v
training/Adam/conv2/bias/v
training/Adam/conv3/kernel/v
training/Adam/conv3/bias/v
training/Adam/dense1/kernel/v
training/Adam/dense1/bias/v
training/Adam/dense2/kernel/v
training/Adam/dense2/bias/v
training/Adam/dense_output/kernel/v
training/Adam/dense_output/bias/v
WARNING:tensorflow:From keras_to_frozen.py:32: convert_variables_to_constants (from tensorflow.python.framework.graph_util_impl) is deprecated and will be removed
 in a future version.
Instructions for updating:
Use `tf.compat.v1.graph_util.convert_variables_to_constants`
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\framework\graph_util_impl.py:277: extract_sub_graph (from tensorflow.
python.framework.graph_util_impl) is deprecated and will be removed in a future version.
Instructions for updating:
Use `tf.compat.v1.graph_util.extract_sub_graph`
WARNING:tensorflow:From keras_to_frozen.py:59: The name tf.train.write_graph is deprecated. Please use tf.io.write_graph instead.
```

Fig 4: Conversion Of Keras to Frozen Graph



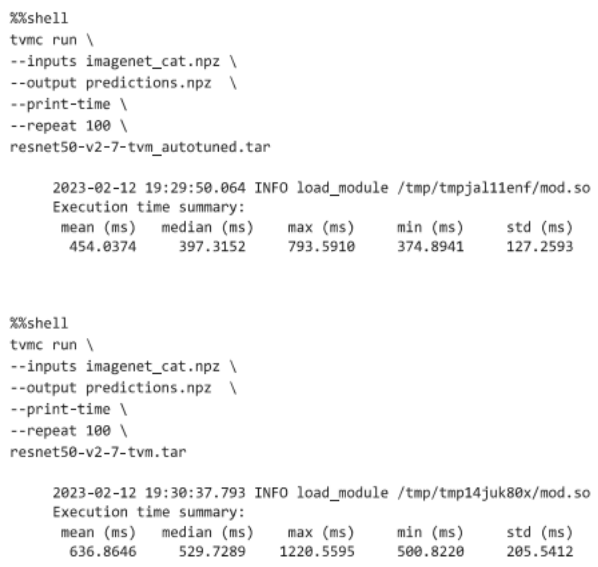
```
(tensorflow) C:\Users\kisho\Desktop\New folder\Conversions>python normal_pb_to_tflite.py
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling VarianceScaling.__init__ (from tensorflow
.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling Zeros.__init__ (from tensorflow.python.op
s.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__ini
t__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling GlorotUniform.__init__ (from tensorflow.p
ython.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
```

Fig 5: Conversion Of Protobuf to Tflite



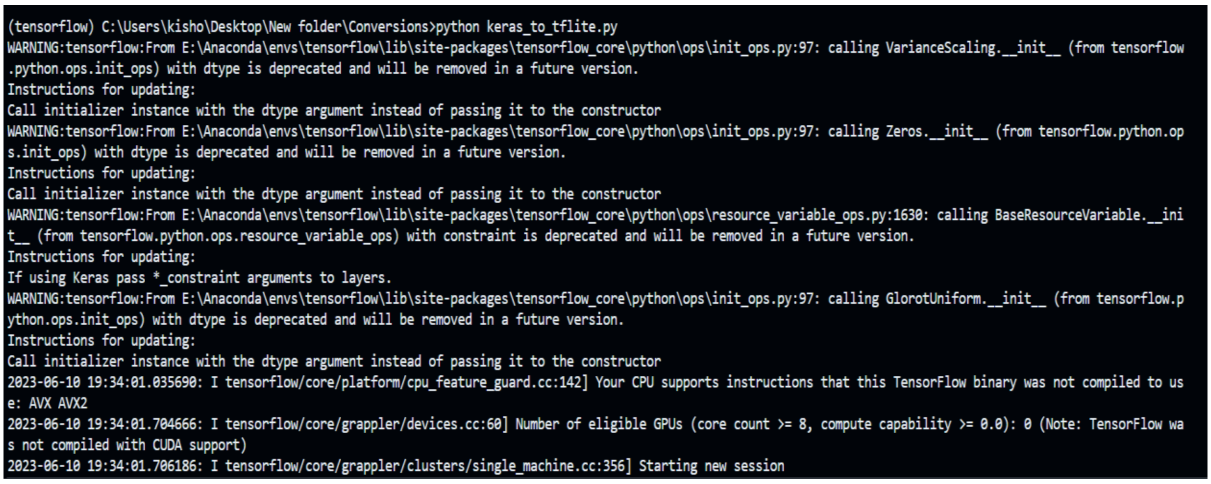
```
(tensorflow) C:\Users\kisho\Desktop\New folder\Conversions>python optimized_pb_to_tflite.py
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling VarianceScaling.__init__ (from tensorflow
.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling Zeros.__init__ (from tensorflow.python.op
s.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__ini
t__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling GlorotUniform.__init__ (from tensorflow.p
ython.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
```

Fig 6: Conversion Of Optimized Protobuf to Tflite



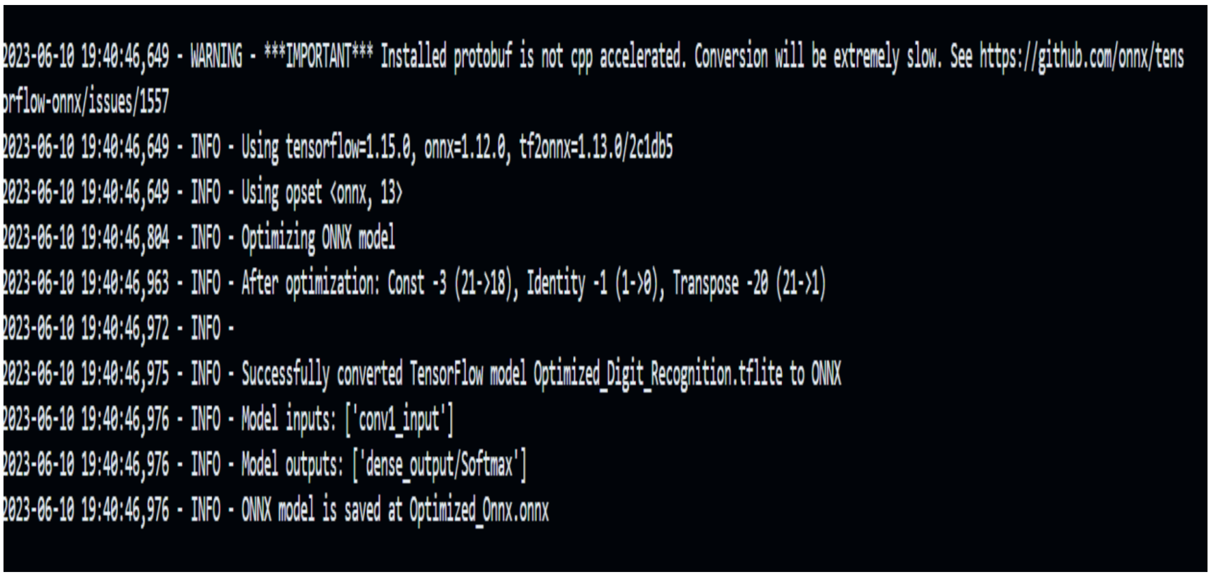
```
%%shell
tvmc run \
--inputs imagenet_cat.npz \
--output predictions.npz  \
--print-time \
--repeat 100 \
resnet50-v2-7-tvm_autotuned.tar

     2023-02-12 19:29:50.064 INFO load_module /tmp/tmpjal1lenf/mod.so
     Execution time summary:
      mean (ms)   median (ms)    max (ms)     min (ms)     std (ms)
       454.0374     397.3152     793.5910     374.8941     127.2593


%%shell
tvmc run \
--inputs imagenet_cat.npz \
--output predictions.npz  \
--print-time \
--repeat 100 \
resnet50-v2-7-tvm.tar

     2023-02-12 19:30:37.793 INFO load_module /tmp/tmp14juk80x/mod.so
     Execution time summary:
      mean (ms)   median (ms)    max (ms)     min (ms)     std (ms)
       636.8646     529.7289    1220.5595     500.8220     205.5412
```

Fig 7: TVM Model Summary



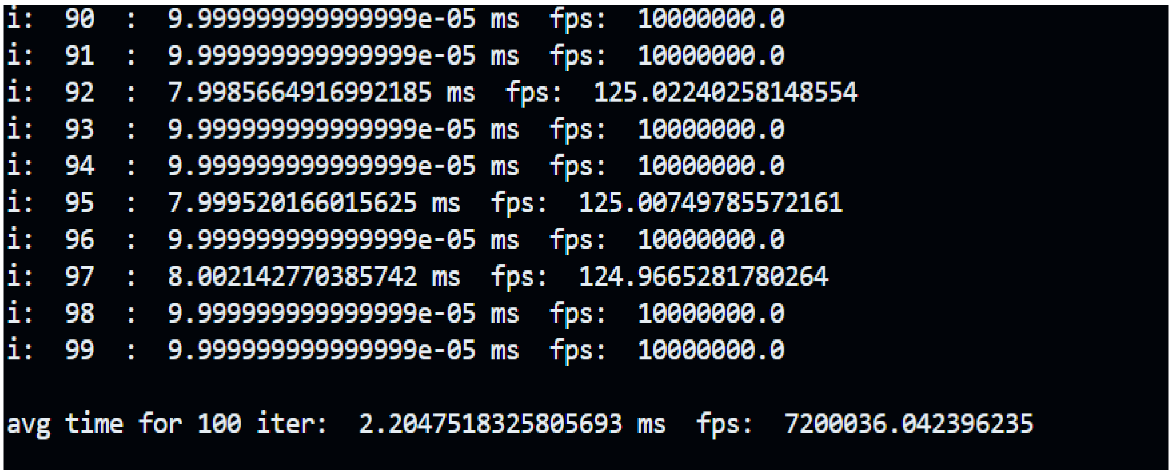
```
(tensorflow) C:\Users\kisho\Desktop\New folder\Conversions>python keras_to_tflite.py
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling VarianceScaling.__init__ (from tensorflow
.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling Zeros.__init__ (from tensorflow.python.op
s.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__ini
t__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
WARNING:tensorflow:From E:\Anaconda\envs\tensorflow\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97: calling GlorotUniform.__init__ (from tensorflow.p
ython.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
2023-06-10 19:34:01.035690: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlow binary was not compiled to us
e: AVX AVX2
2023-06-10 19:34:01.704666: I tensorflow/core/grappler/devices.cc:60] Number of eligible GPUs (core count >= 8, compute capability >= 0.0): 0 (Note: TensorFlow wa
s not compiled with CUDA support)
2023-06-10 19:34:01.706186: I tensorflow/core/grappler/clusters/single_machine.cc:356] Starting new session
```

Fig 8: Conversion Of Keras to Tflite



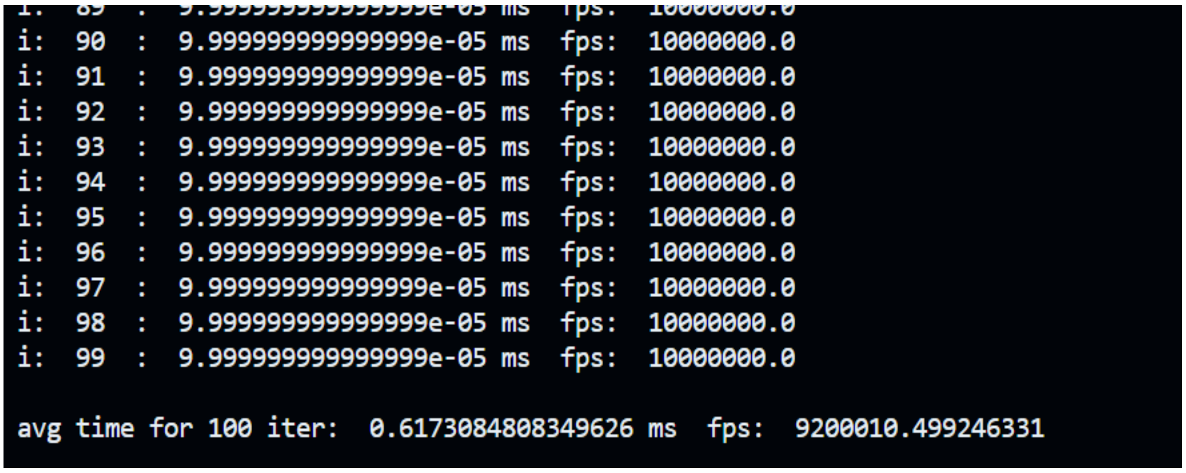
```
023-06-10 19:40:46,649 - WARNING - ***IMPORTANT*** Installed protobuf is not cpp accelerated. Conversion will be extremely slow. See https://github.com/onnx/tens
rflow-onnx/issues/1557
023-06-10 19:40:46,649 - INFO - Using tensorflow=1.15.0, onnx=1.12.0, tf2onnx=1.13.0/2c1db5
023-06-10 19:40:46,649 - INFO - Using opset <onnx, 13>
023-06-10 19:40:46,804 - INFO - Optimizing ONNX model
023-06-10 19:40:46,963 - INFO - After optimization: Const -3 (21->18), Identity -1 (1->0), Transpose -20 (21->1)
023-06-10 19:40:46,972 - INFO -
023-06-10 19:40:46,975 - INFO - Successfully converted TensorFlow model Optimized_Digit_Recognition.tflite to ONNX
023-06-10 19:40:46,976 - INFO - Model inputs: ['conv1_input']
023-06-10 19:40:46,976 - INFO - Model outputs: ['dense_output/Softmax']
023-06-10 19:40:46,976 - INFO - ONNX model is saved at Optimized_Onnx.onnx
```

Fig 9: Conversion Of Tflite to Onnx



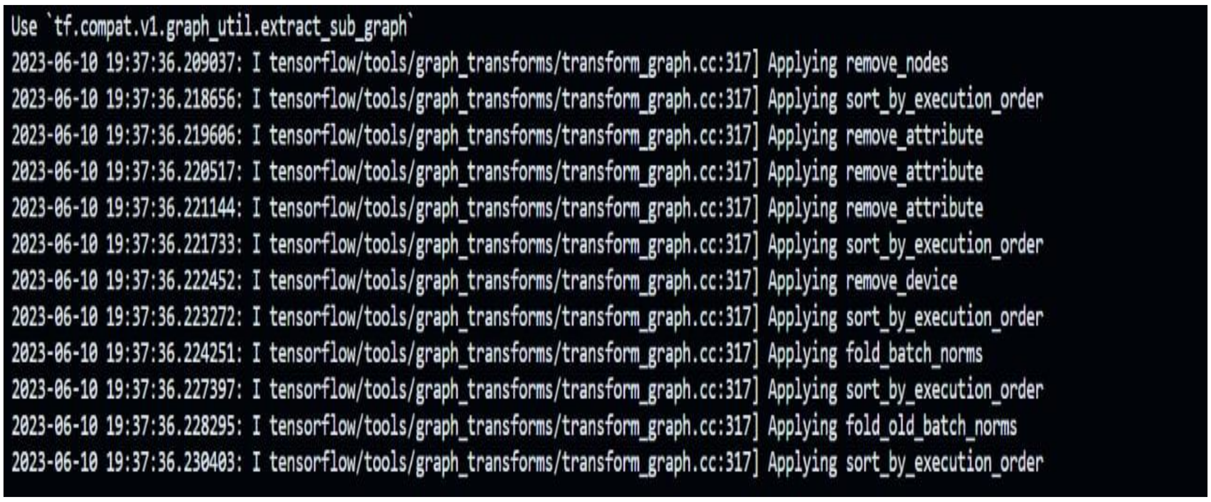
```
i:  90  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  91  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  92  :  7.9985664916992185 ms  fps:  125.02240258148554
i:  93  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  94  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  95  :  7.999520166015625 ms  fps:  125.00749785572161
i:  96  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  97  :  8.002142770385742 ms  fps:  124.9665281780264
i:  98  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  99  :  9.999999999999999e-05 ms  fps:  10000000.0

avg time for 100 iter:  2.2047518325805693 ms  fps:  7200036.042396235
```

Fig 10: Frozen Graph Evalution



```
i:  89  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  90  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  91  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  92  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  93  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  94  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  95  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  96  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  97  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  98  :  9.999999999999999e-05 ms  fps:  10000000.0
i:  99  :  9.999999999999999e-05 ms  fps:  10000000.0

avg time for 100 iter:  0.6173084808349626 ms  fps:  9200010.499246331
```

Fig 11: Onnx Evalution



```
Use `tf.compat.v1.graph_util.extract_sub_graph`
2023-06-10 19:37:36.209037: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying remove_nodes
2023-06-10 19:37:36.218656: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying sort_by_execution_order
2023-06-10 19:37:36.219606: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying remove_attribute
2023-06-10 19:37:36.220517: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying remove_attribute
2023-06-10 19:37:36.221144: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying remove_attribute
2023-06-10 19:37:36.221733: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying sort_by_execution_order
2023-06-10 19:37:36.222452: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying remove_device
2023-06-10 19:37:36.223272: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying sort_by_execution_order
2023-06-10 19:37:36.224251: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying fold_batch_norms
2023-06-10 19:37:36.227397: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying sort_by_execution_order
2023-06-10 19:37:36.228295: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying fold_old_batch_norms
2023-06-10 19:37:36.230403: I tensorflow/tools/graph_transforms/transform_graph.cc:317] Applying sort_by_execution_order
```

Fig 12: Optimized Model

7. Future Scope

a. **Explainability and interpretability:** As deep learning models become more complicated, it becomes more difficult to grasp how they make decisions. Researchers are focusing on approaches to increase the explainability and interpretability of deep learning models.
b. **AutoML and Hyperparameter Tuning:** AutoML is a new field that employs machine learning to automate the creation of machine learning models. More powerful AutoML algorithms that can autonomously optimise neural network structures and hyperparameters are expected in the future.
c. **Distributed and Federated Learning:** Large-scale deep learning models require a significant amount of computational power, and training them can take a long time. Distributed and federated learning can help reduce training times by distributing the workload across multiple machines or devices.

8. Applications

Optimized model can be used in wide range of domains

1. **OCR**:
   Optimized models for Optical Character Recognition (OCR) surpass normal non-optimized models by delivering higher accuracy, faster inference speed, improved efficiency, enhanced robustness, and suitability for resource-constrained devices. These models leverage techniques such as attention mechanisms, data augmentation, quantization, and hardware acceleration to achieve superior performance. With optimized OCR models, businesses can benefit from more accurate text recognition, real-time processing capabilities, cost-effective scalability, and streamlined workflows in applications such as document digitization, text extraction, language translation, accessibility tools, and fraud detection.

2. **Autonomous Vehicles:**
   Optimized DNN models play a crucial role in autonomous vehicles system, which capable of operating without human mediation. These vehicles rely on various sensors, such as cameras and LiDAR, to perceive the environment and make decisions in real time. The DNN models used in autonomous vehicles are responsible for tasks such as object detection, lane detection, traffic sign recognition, and pedestrian detection.

3. **Empowering Camera Efficiency & Performance:**
   Optimized models offer significant advantages over non-optimized models, making them more effective and beneficial in the context of cameras. Firstly, optimized models are designed to maximize computational efficiency, utilizing advanced algorithms and hardware optimizations. This results in faster processing times, enabling real-time object detection, tracking, and other computer vision tasks. Non-optimized models may struggle to meet the demanding real-time requirements of camera applications.

   Secondly, optimized models are built to minimize power consumption. Through techniques like model compression, pruning, and quantization, they reduce the computational and memory requirements of the model. This is particularly crucial for cameras operating on battery power or in resource-constrained environments. By consuming less power during inference, optimized models can extend the camera's battery life and improve energy efficiency.

   Additionally, optimization techniques enhance the accuracy of models. By reducing overfitting and improving generalization, optimized models achieve higher accuracy in tasks such as object detection, recognition, and tracking. They can better handle variations in lighting conditions, object sizes, and complex scenes, providing more reliable and precise results compared to non-optimized models.

   Furthermore, optimized models prioritize efficient memory usage. Techniques like weight pruning and quantization reduce the memory footprint of the model, allowing optimized models to run on cameras with limited memory resources. This makes them suitable for deployment in resource-constrained camera systems.

Lastly, optimized models can be tailored for specific camera tasks. By fine-tuning or customizing the model architecture and parameters, they can achieve superior performance in areas such as face recognition, gesture detection, or scene understanding. Non-optimized models may lack this task-specific customization, resulting in suboptimal performance for specific camera applications.

In conclusion, optimized models in cameras offer faster processing, reduced power consumption, improved accuracy, efficient memory usage, and task-specific customization. These advantages enhance camera performance, enable real-time analysis, extend battery life, and ensure more accurate and reliable results. Optimized models are crucial for maximizing the capabilities of cameras in various domains, including surveillance, autonomous vehicles, and smart devices.

4. **Healthcare Diagnosis:**

Optimized DNN models can be highly valuable in healthcare diagnosis, where accurate and timely detection of diseases can have a significant impact on patient outcomes. These models can aid healthcare professionals in diagnosing various medical conditions based on patient data, such as medical images, electronic health records, and clinical notes.

9. Plagiarism Report

Document Viewer
Similarity Index

7%
test
By: test
As of: Jun 9, 2023 5:27:41 PM 6,528 words - 40 matches - 17 sources
sources:
41 words / 1% - Internet from 05-Apr-2020 12:00AM
www.slideshare.net
19 words / < 1% match - Internet from 26-Jan-2023 12:00AM
www.slideshare.net
13 words / < 1% match - Internet from 21-Dec-2022 12:00AM
www.slideshare.net
32 words / < 1% match - from 04-Apr-2023 12:00AM
WWW.coursehero.com
11 words / < 1% match - Internet from 28-Feb-2021 12:00AM
www.coursehero.com
21 words / < 1% match - Internet from 07-Feb-2023 12:00AM
www.researchgate.net
14 words / < 1% match - Internet from 02-Feb-2023 12:00AM
www.researchgate.net
19 words / < 1% match - Internet from 30-Dec-2022 12:00AM
www.scribd.com
9 words / < 1% match - Internet from 22-Apr-2020 12:00AM
www.scribd.com

19 words / < 1% match - Internet from 29-Nov-2022 12:00AM
www.mdpi.com
18 words / < 1% match - Internet from 07-Feb-2019 12:00AM
textilevaluechain.com
18 words / < 1% match - from 26-May-2023 12:00AM
www.irjmets.com

9 words / < 1% match - Internet from 25-Jan-2023 12:00AM
limbimoderne.lls.unibuc.ro

8 words / < 1% match - Internet from 13-Aug-2022 12:00AM
careersdocbox.com
8 words / < 1% match - Internet from 23-Nov-2022 12:00AM
dspace.dtu.ac.in
8 words / < 1% match - from 28-May-2023 12:00AM
ijircce.com
8 words / < 1% match - from 23-May-2023 12:00AM
onshow.iadt.ie

10.References

1. Hanan Hussain "Design possibilities and challenges of DNN models: a review on the perspective of end devices," Springer Nature,16 January 2022
2. Peemen, M., Setio, A. A. A., Mesman, B., & Corporaal, H. (2013). "Memory-centric accelerator design for Convolutional Neural Networks". 2013 IEEE 31st International Conference on Computer Design (ICCD).
3. Qingchao Shen, "A Comprehensive Study of Deep Learning Compiler Bugs", ACM Conferences, 4, 18 August 2021.
4. Arvind Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning", Carlsbad CA US, 579–594, 08 October 2018.
5. Ananthram Swami, "The Limitations of Deep Learning in Adversarial Settings", 2016 IEEE European Symposium on Security and Privacy (Euro S&P), 12 May 2016.
6. Pavel Kordik, "Meta Learning approach to Neural Network Optimization", The 18th International Conference on Artificial Neural Networks, ICANN 2008, Vol-23, 578, May 2010.
7. Marchisio, A., Hanif, M. A., Khalid, F., Plastiras, G., Kyrkou, C., Theocharides, T., & Shafique, M. (2019). "Deep Learning for Edge Computing: Current Trends, Cross-Layer Optimizations, and Open Research Challenges." 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI).
8. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
9. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Proceedings of the 32nd International Conference on Machine Learning (ICML) (Vol. 37, pp. 448-456).
10. Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.